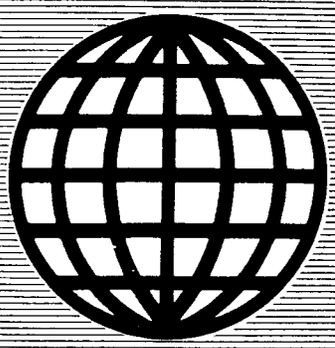


The COMPUTER JOURNAL®

Programming - User Support Applications

Issue #27

$3.00

68000 TinyGiant

Low Cost 16-bit SBC and Operating System

The Art of Source Code Generation

Disassembling Z-80 Software

Feedback Control System Analysis

Using Root Locus Analysis and Feedback Loop Compensation

The Hitachi HD64180

New Life for 8-bit Systems

A Tutor Program for Forth

Writing a Forth Tutor in Forth

Disk Parameters

Modifying The CP/M Disk Parameter Block for Foreign Disk Formats

ISSN # 0748-9331

THE COMPUTER JOURNAL
190 Sullivan Crossroad
Columbia Falls, Montana
59912
406-257-9119

Editor/Publisher
Art Carlson

Art Director
Donna Carlson

Production Assistant
Judie Overbeek

Circulation
Donna Carlson

Contributing Editors
C.Thomas Hilton
Donald Howes
Jerry Houston
Bill Kibler
Rick Lehrbaum
Peter Ruber
Jay Sage
Jon Schneider

Entire contents copyright© 1987 by The Computer Journal.

***Subscription rates**—$16 one year (6 issues), or $28 two years (12 issues) in the U.S., $22 one year in Canada and Mexico, and $24 (surface) for one year in other countries. All funds must be in US dollars on a US bank.*

Send subscriptions, renewals, or address changes to: The Computer Journal, 190 Sullivan Crossroad, Columbia Falls, Montana, 59912, or The Computer Journal, PO Box 1697, Kalispell, MT 59903.

***Bulletin Board**—Our bulletin board will be on line 24 hours a day at 300 and 1200 baud, and the number is (406) 752-1038.*

Address all editorial and advertising inquiries to: The Computer Journal, 190 Sullivan Crossroad, Columbia Falls, MT 59912 phone (406) 257-9119.

The COMPUTER JOURNAL

Features

68000 TinyGiant
Hawthorne's 68000 Single Board Computer combined with their $50 operating system
by Art Carlson. 4

The Art of Source Code Generation
Disassembling Z-80 software to produce modifiable source code
by Clark A. Calkins. 8

Feedback Control System Analysis
Using Root Locus analysis and Feedback Loop Compensation to solve closed loop control problems
by Bert P. van den Berg. 14

The Hitachi HD64180
Understanding the HD64180's advantages over the Z-80, and how to use some of it's advanced features
by Jon Schneider. 23

A Tutor Program for Forth
Writing a Forth Tutor in Forth
by Bill Kibler. 36

Disk Parameters
Modifying the CP/M Disk Parameter Block to read and write foreign disk formats
by C. Thomas Hilton. 40

Departments

Editorial 2

C Corner
by Donald Howes. 18

ZSIG Corner
by Jay Sage. 28

Reader's Feedback 34

Computer Corner
by Bill Kibler. 52

Editor's Page

Market Developments

While some people claim that the personal computer sales are flat because the sales of their own products are low, and others claim that the technology is stagnate because all they see is the IBM PC series, there is actually a great deal of activity in both sales of existing products and technical developments which will lead to the next generation of computers.

If anyone tells you that computers aren't selling, show them the current issue of Computer Shopper which has 408 pages mostly filled with ads for hardware and software. Many of these advertisers have been running full page or even multipage ads in every issue, and those I talk to assure me that the ads are generating the sales to justify the cost of the ads — they're selling a lot of hardware, software, and peripherals — but the market has made another abrupt change. As pointed out in George Morrow's column in Electronic Engineering Times (January 12, 1987, page 61), individuals started buying again in 1986 while the corporations and government agencies cut way back on their purchases. To quote one small section of his column, "The personal computer market was launched by individuals who were convinced that the power of low-cost micros could be harnessed to solve spreadsheet applications and word processing chores, and it's no secret that this type of customer doesn't use the same criteria as corporations or government agencies when it comes to buying computers. When a government agency buys a computer, the taxpayer pays the bill. When a corporation buys a computer, the company's stockholders pay for it. Individuals, however, have no choice but to spend their own money. So the price of the machine, as well as its features (clock rate, ease of use, bundled software, etc.) become deciding factors in a sale." George goes on to say, "Thus, the key to next year seems to be in the hands of individuals, just as it was in 1986. Will they continue to buy at the pace they established in 1986?"

Digesting what George said, I feel that when corporations and government agencies became the primary buyers, most sellers structured their product line, marketing organization, and pricing for this 'big-business' segment. Now that individuals and small companies are again the primary buyers, these sellers have too much built-in overhead cost and are burdened with products and marketing plans designed for those spending someone else's money. The ones succeeding in the current market are the lean and hungry ones who can provide the features and prices demanded by the individuals spending their own money.

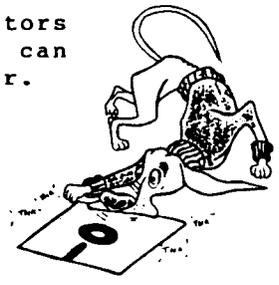
At the same time as the current products are being sold in volume, there are also a lot of technical developments taking place which will affect what we buy in the near future — but these developments are not taking place in the consumer market which everyone is watching, but rather in the industrial market where they are not attracting much attention.

There is a lot of talk about how much power the Motorola 68020, 68030, and 68040 and the Intel 80286, and 80386 will provide, but there is very little talk about what kind of system architecture will be needed in order to realize these benefits. It's true that MicroSoft and others are developing operating systems for the 80386, but apparently the hardware people (remember that they are used to dealing with buyers spending some one else's money) are talking about putting it on a glorified PC-type bus. That's like putting a turbo-charged V-12 engine in a old Ford model T.

I feel that the advanced business, engineering, and industrial systems which can properly utilize the power of the new CPU's will be multiuser, multitasking systems employing mutiprocessors and will require a

Ever Wondered What Makes CP/M® Tick?

Source Code Generators by C. C. Software can give you the answer.



"The darndest thing I ever did see..."
"... if you're at all interested in what's going on in your system, it's worth it."
Jerry Pournelle, BYTE, Sept '83

The S.C.G. programs produce fully commented and labeled source code for your CP/M system (the CCP and BDOS areas). To modify the system to your liking, just edit and assemble with ASM. CP/M 2.2 $45, CP/M+ $75, + $1.50 postage (in Calif add 6.5%).

C. C. Software, 1907 Alvarado Ave.
Walnut Creek, CA 94596 (415)939-8153

CP/M is a registered trademark of Digital Research, Inc.

reliable bus structure which provides for interprocessor communications and multiprocessor communications. I'm sure that many readers will comment on my negative stand on the current multitasking and multiuser implementations — and that's because I feel that the currrent systems do not have sufficient power to properly execute these feature. If we don't need these features, then we don't need the new chips either! I'm happy with my 8-bit systems with hard disks for wordprocessing which is the majority of my work. I'll probably upgrade to solid state RAM to speed things up a little and avoid the disk noise, but I surely don't need a 16-bit or 32-bit CPU and I don't need multiuser or multitasking for our operation (although other larger operations may well need this). Eight bit systems are cheap, and if I want to do several things at the same time, I just use several systems at the same time!

I am very interested in the new chips for other applications, but when I choose a powerful chip because of its features I **insist on being able to use ALL of its speed and power!** If limited power is acceptable, I can provide that with existing CP/M or PC-DOS equipment. If more power is needed, then I want to be able to use everyting that I can get.

The industrial market has already developed architectures such as the VMEbus® (Motorola) and Multibus II® (Intel) which provide most of the features required to use the full power of the new CPUs, and 68020, 80286, 80386, and WE32XXX single board computers are readily available. Simple low power systems can be used with a simple low power bus — or even without a bus — but high-power systems must be used with a high-power bus in order to realize their full potentials and to provide for expansion. I'll only be interested in using the new CPUs on a standard open bus, except possibly for a dedicated single use application such as a CAD station.

While we're talking about the new CPU's, it appears that the software developers have stopped their work on the AT 80286 in favor of the 80386. Without the software to make use of the '286's features, the AT will just be used as a fast PC instead of realizing its full potential.

Why Companies Don't Succeed

Most companies want to have their products considered for new designs, and they spend a lot of money for slick four color ads, but then when we try to get technical information they fall flat on their face. An example is the AD639 Trig Processor chip from Analog Devices. I wrote three letters asking for tech info to publish, but never received a response. In November I finally phoned and talked to someone in marketing who promised to send the info. We still haven't received anything, and frankly I'm no longer interested in dealing with them.

On the other hand, I called Hitachi for info on their HD64180 and received a large box of material. They also followed up with a phone call to see if I had received it and if I needed any further help.

Guess which product I'll support. The American companies will continue to cry about how they are losing business to the foreign producers, but in many cases it is their own fault because while they spend money on impressive advertising they won't pay attention to customers and provide support.

Hard Drives vs. RAM Disks

Hard drives are nice for operations involving a lot of disk access such as data bases or compiling where the speed and capacity advantages over floppies are important — BUT remember that when used continuously, every hard drive will eventually FAIL! The question is not IF, but rather WHEN. The expensive large drives intended for mainframe applications will probably last for a long time, but some of the low cost drives for micros are not so durable. In the case of the 10 Meg drive for our BBS the bearings gave out after about six months of continuous operation. I noticed it in time to save the data, but I'll have to send the drive out for repair.

The board is running on just two floppies until I decide on a long term solution, but I question if I want a hard drive running 24 hours a day for seven days a week. I may add more 96TPI floppies because the drives only run when they are accessed instead running all the time like a hard drive. The real solution will be to go to battery backed RAM disk if and when I can justify the cost. I would be very interested in feedback regarding other people's experience with the life of various models of hard drives.

Source Code

I realize that you get tired of hearing me talk about source code, but having the source code is what enables us to

(Continued on page 38)

Make certain that TCJ follows you to your new address. Send both old and new address along with your expiration number that appears on your mailing label to:

THE COMPUTER JOURNAL
190 Sullivan Crossroad
Columbia Falls, MT 59912

Please allow six weeks notice. Thanks.

Registered Trademarks

It is easy to get in the habit of using company trademarks as generic terms, but these registered trademarks are the property of the respective companies. It is important to acknowledge these trademarks as their property to avoid their losing the rights and the term becoming public property. The following frequently used marks are acknowledged, and we apologize for any we have overlooked.

Apple II, II + , IIc, IIe, Macintosch, DOS 3.3, ProDOS; Apple Computer Company. CP/M, DDT, ASM, STAT, PIP; Digital Research. MBASIC; Microsoft. Wordstar; MicroPro International Corp. IBM-PC, XT, and AT; IBM Corporation. Z-80, Zilog. MT-BASIC, Softaid, Inc. Turbo Pascal, Borland International.

Where these terms (and others) are used in The Computer Journal they are acknowledged to be the property of the respective companies even if not specifically mentioned in each occurence.

68000 TinyGiant

Low Cost 16-bit SBC and Operating System

by Art Carlson

There has been a lot of interest in a low priced 68000® single board computer for experimenting or process control, and a group of us got together at SOG last year to discuss the situation. We are interested in the 68000 because it is big, fast, cheap, and well supported. The 86 family is dominated by MS-DOS/PC-DOS® machines which are designed as office machines, and the other 32 bit micro processors are not very well supported. We wanted to work with a processor for which books are available, and we also wanted other companies to provide software for the new machine.

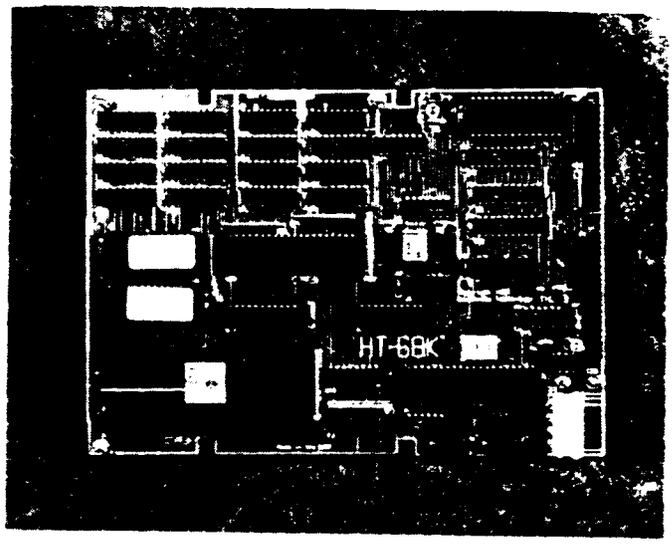
The major problem in experimenting with the 68000 has been the lack of a readily available low-cost operating system and programming utilities. As a result of our discussions, Joe Bartel (Hawthorne Technology) offered to supply a single-user version of his 68000 operating system complete with the source code and the compiler for his HTPL® language (in which the system is written) for $50. He's even throwing in a line editor and a 68000 assembler.

Once we had the operating system, we started searching for suitable hardware since not all of us are interested in designing and wirewrapping the main CPU board. I understand that some people are porting Hawthorne's K-OS ONE® operating system over to the Atari ST® (which is a great hardware bargain), but we also need a small low cost board which can be mounted on robotic arms and other places where an ST is too big or is technically unsuitable. After all, for many controller applications we don't need sound, graphics, drives, keyboard, or monitor; but we do need complete bus access so that we can provide lots of I/O. Joe came up with another winner which is the 68000 single board computer he calls "The TinyGiant® ."

Every computer is designed to fill a particular goal or market niche, and The TinyGiant was designed to be as low cost as possible while still providing the needed features. The target price was set at $395, and when a choice had to be made they looked at the original goal. Then they looked at what was needed to make a usable small developement system. There had to be a disk of some kind. There also had to be enough I/O to support a terminal and some kind of printer and a modem. They didn't put a video controller on the board because they felt that many of these would be used in places where video was not needed. Also video would have made the board much larger and more costly. Dynamic RAMs were used because of the lower cost.

The TinyGiant is a complete 68000 system on a single board. The main component is the 68000 that runs at 8 Mhz. with no wait states. There are two EPROMS, a dynamic RAM controller, four RAM chips, and a 68681® dual serial chip. There is also an unbuffered expansion bus provided.

The 68681 is almost a complete I/O system on a single chip. There are two serial ports and a timer on the 68681. There are also some extra input and output bits that can be used. Each serial port has its own internal baud rate generator that can be set from 50 to 9600 baud. The system startup assumes that the console terminal on the first port is set to 9600 baud. The other port is assumed to be for a modem and is set to 1200 baud. This port could be used to support a serial printer if a modem is not needed. The timer is used as the basic system timer for a time of day clock. When the PRN device is selected, a parallel printer port made from a latched buffer is used.



68000 TinyGiant

The 1770® floppy controller can control up to four 5¼ or 3½ inch floppy disk drives. At the present time the software is only supplied on 5¼ inch floppy disks, but it will also be supplied on 3½ disks in the near future. The 1770 controller has an internal clocked digital data separater so no adjustments are needed. There is no DMA controller so all I/O is done with interrupts.

The board is supplied with 128k of RAM but can be expanded to 512k by simply adding the memory chips. The memory is controlled with an MB1422® dynamic RAM controller which takes care of all the timing and refresh for the 64k x 4 dynamic RAMS. There are four RAM chips in the basic machine, 12 more can be added on board, and an expansion board can be added to provide additional memory. There are two EPROMS that contain the boot code and a very simple hardware debug monitor, and there is a lot of empty space in the PROMS that could later be used for code.

There are two expansion connectors that provide all the needed address, data, and control lines. The 64 pin expansion bus connector has all the raw signals just as they are on the 68000 itself and it is an unbuffered bus which means that buffers have to be provided for the bus lines on the first expansion board. The other expansion connector has some signals derived from logic on board. There is a priority interrupt encoder that is

connected to autovector interrupts.

To use the TinyGiant you have to connect a power supply, a terminal, and a disk drive. The board requires only 5 volts at about 2 amp and some at +12 volts for the RS-232 (the −12 for the RS-232 is generated on board), and the power connector is the same as a floppy drive. The EPROM monitor is set up to boot the K-OS ONE operating system from a floppy disk when the power is applied. All of this should be in a proper case, and any of the cases for the Ampro Little Board® should work fine because the board is the same size. The cases and power supplies for the PC clones should also work and are available at very low prices.

The software that comes with the TinyGiant has many features that will remind you of CP/M® or MS-DOS. The disks are compatible with MS-DOS which means that you can move data files from your new machine to your PC easily. The standard command processor is modeled after MS-DOS but does not have batch files. The editor is a line editor but is much better than ED® or EDLIN® . The assembler is a two pass absolute assembler that has include files but no macros. The compiler is for HTPL, the language used to write the K-OS ONE operating system.

People who currently use a Z-80® and are interested in moving up will be interested in the board. Also anyone who wants to learn about the 68000 without having to dig through a difficult operating system will like it. The price of the board is competitive with the 80 and 86 family boards and so is the price of the software. The small size and power make it great for control applications. The use of MS-DOS compatible disks makes it very convenient for data capture. The operating system makes it good for a personal computer or for experimenting.

At the present time there are a limited number of programs that will run under K-OS ONE. This will improve in the next few months as the companies that purchased the first copies of K-OS ONE bring their products to market. For any new system there is a delay between when it is first delivered and when lots of software is available for it. Some of the software that will be available is code that is being ported from other systems.

For languages there is the assembler and the HTPL compiler that come with the basic machine, and the operating system is written in HTPL so that it can be recompiled on the system. Most of the development is being done in HTPL, but the source code for the runtime library is written in assembler and is supplied in source form. For many applications most of the code can be written in HTPL with the speed critical portions written in assembler. Also there are some functions that are much easier to write or much smaller in assembler. I understand that a C compiler is being ported over to the system.

Externally, the operating system can look like anything you want it to. The command processor provided looks a lot like MS-DOS but you can change it because it is a separate program like the CCP is in CP/M and you are provided with the source code. In a dedicated application the command processor could be replaced by a menu to load a set of application programs. A command processor could also be written that would make it look like a Unix system was being used if that was desired.

Internally, the operating system resembles Unix® or MS-DOS 2.0 except that it is greatly simplified. All system calls are done in a manner similar to MS-DOS where the required information is placed in a parameter block and then a trap instruction gives control to the system. All error information is returned in memory in the parameter block. This makes it very easy to use system fuctions directly in a high level language. The use of Unix like files instead of file control blocks like CP/M means that any application written will still run even if there are radical changes to the internal structure of the disks.

HT68K TinyGiant Hardware Description

There are 6 connectors on the TinyGiant board that connect it to the real world. Two of these are for expansion, one for the console serial port, one for the auxillary serial port, one for the printer, and finally one for the disk drives.

The size and shape of the board is such that it can be bolted to the side of a standard 5¼ inch floppy disk drive, and the power connector for the +5 and +12 is the same as used on a floppy disk drive. The +12 volts is converted to −12 for RS-232 by an onboard power converter. Most of the cheap PC clone power supplies already have the connectors wired for floppy disks so one of them can be used.

The expansion bus is divided between two connectors, P1 and P2. The P1 connector brings out all the basic 68000 control signals. These come directly from the 68000 cpu chip and are not buffered. The remaining signals come from the P2 connector. Any signal should be buffered before it is used on an expansion board.

When power is applied the PROM is at location 0. As soon as the serial port is addressed the PROM is shifted into high memory. This is taken care of in the boot PROM so all the user will see is RAM memory at location 0. The BIOS is in PROM so it is assumed to run at $380000. All interrupts are auto vectored, some are dedicated to devices on the board and some are available on the expansion connector.

P1 CONNECTOR -- EXPANSION

1	D4	33	D5
2	D3	34	D6
3	D2	35	D7
4	D1	36	D8
5	D0	37	D9
6	AS*	38	D10
7	UDS*	39	D11
8	LDS*	40	D12
9	R/W*	41	D13
10	DTACK*	42	D14
11	BG*	43	D15
12	BGACK*	44	GND
13	BR*	45	A23
14	VCC, 5 V	46	A22
15	CLK, 8 MHZ	47	A21
16	GND	48	VCC, 5V
17	HALT*	49	A20
18	RESET*	50	A19
19	VMA*	51	A18
20	E	52	A17
21	VPA*	53	A16
22	BERR*	54	A15
23	IPL2*	55	A14
24	IPL1*	56	A13
25	IPL0*	57	A12
26	FC2	58	A11
27	FC1	59	A10
28	FC0	60	A9
29	A1	61	A8
30	A2	62	A7
31	A3	63	A6
32	A4	64	A5

```
P2 CONNECTOR -- EXPANSION

 1 = EXIO2*          9 = EXIO1*
 2 = +12            10 = DREQ*
 3 = -12            11 = NMI*
 4 = VCC, 5 V       12 = EXIR1*
 5 = VCC            13 = EXIR5*
 6 = GND            14 = EXIR6*
 7 = GND            15 = GND
 8 = GND            16 = 16 MHZ CLOCK
P3 CONNECTOR -- DISK

 1 = GND             2 = NC
 3 = GND             4 = NC
 5 = GND             6 = DSO
 7 = GND             8 = IP
 9 = GND            10 = DS3
11 = GND            12 = DS2
13 = GND            14 = DS1
15 = GND            16 = MO
17 = GND            18 = DIR
19 = GND            20 = STEP
21 = GND            22 = WD
23 = GND            24 = WG
25 = GND            26 = TKO
27 = GND            28 = WP
29 = GND            30 = RD
31 = GND            32 = SIDE SEL
33 = GND            34 = NC

P4 CONNECTOR -- TERMINAL SERIAL CONNECTOR
 1 = GND             6 = NC
 2 = RXD             7 = GND
 3 = TXD             8 = NC
 4 = CTS             9 = NC
 5 = RTS            10 = NC

P4 CONNECTOR -- MODEM SERIAL CONNECTOR
 1 = GND             6 = NC
 2 = RXD             7 = GND
 3 = TXD             8 = DCD
 4 = CTS             9 = NC
 5 = RTS            10 = RTS

PR CONNECTOR -- PRINTER
 1 = STROBE
 2 = D0
 3 = D1
 4 = D2
 5 = D3
 6 = D4
 7 = D5
 8 = D6
 9 = D7
10 = ACK -- BUSY

MEMORY MAP

$000000 - $07FFFF   RAM ON BOARD
$080000 - $08FFFF   X1
$100000 - $10FFFF   X2
$180000 - $17FFFF   PRT
$200000 - $27FFFF   SLK- FDC
$280000 - $2FFFFF   SIO
$300000 - $37FFFF   FDC
$380000 - $3FFFFF   PROM ON BOARD

INTERRUPT LEVELS
0 = not an interrupt at all
1 = external
2 = Printer, internal
3 = Serial, internal
4 = Floppy disk, internal
5 = external
6 = external
7 = NMI, external
```

Where Do We Go From Here?

It's very likely that the 68000 will replace the Z80 as the choice for hobby and experimental projects, and the combination of the TinyGiant and the K-OS ONE operating system will provide a very good development platform.

The lack of software is a major problem with any new hardware or operating system — developers won't write the programs until a lot of the systems are sold, and people won't buy the systems until a lot of software exists — at least that's true with a mass market appliance type computer. But this is initially aimed at a different market, and I believe that Hawthorne has made several very smart moves. For one, they're supplying the system with the board, and the system (whether obtained separately or with the board) includes the OS source code plus the language compiler, a 68000 assembler, and a line editor so that you have the utilities to start writing programs. Another very good move is that even though Hawthorne is selling a single board computer, they still want to sell their operating system to hardware companies even if they are directly competing with Hawthorne! Joe has also been in contact with a lot of potential 68000 programers who were frustrated with the lack of a low cost operating system, and there will soon be a rapidly expanding base of software.

One of the most significant points is that you get the operating system with the source code (plus the other goodies) for only $50. Has any one else ever supplied the OS source code? I doubt it, because many CP/M hardware vendors wouldn't even supply the source code for their customized CBIOS. With the low cost OS and its source, you can customize it for your own particular application, and it's cheap enough so that you can include it with your software.

I feel that the use of K-OS ONE is going to expand very rapidly, and we at TCJ are going to fully support this growth. ■

68000 Reference Book List

68000 16/32-bit Microprocessor Programmer's Reference Manual, by Motorola, Prentice Hall

68000, 68010, 68020 Primer, by Kelly-Bootle and Fowler, The Waite Group, Howard W. Sams

The 68000: Principles and Programming, by Leo Scanlon, Howard W. Sams

68000 Assembly Language: Techniques for Building Programs, by Donald Krantz and James Stanley, Addison-Wesley Publishing Company

68000 Assembly Language Programming, by Lance Leventhal, Doug Hawkins, Gerry Kane and William D. Cramer, Osborne/McGraw-Hill

The 68000 Microprocessor: Architecture, Software and Interfacing Techniques, by W. Triebel, and A. Singh, Prentice Hall

68000 Microprocessor Handbook, by William Cramer, AVM Systems, Gerry Kane, Osborne/McGraw-Hill

68000 User's Manual, by J. Carr, Prentice Hall

MC68020 32 bit Microprocessor User's Manual, by Motorola, Prentice Hall

Basic Microprocessors and the 68000, by Ron Bishop & the Motorola Semiconductor Group, Hayden Books

Dr. Dobb's Toolbook of 68000 Programming, Edited, Prentice Hall

Programming the 68000, by Rosenzweig and Harrison, Hayden Books

Programming the 68000, by Steve Williams, Sybex Computer Books

Self Guided Tour through the 68000, by M. Andrews, Prentice Hall

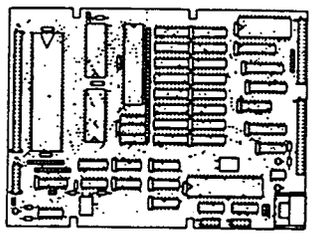
The above books can be ordered from:

Magrathea
8842 Southeast Stark
Portland, OR 97216
Phone (503) 254-2005

68000 SINGLE BOARD COMPUTER

$395.00

32 bit Features / 8 bit Price



-Hardware features:
* 8MHZ 68000 CPU
* 1770 Floppy Controller
* 2 Serial Ports (68681)
* General Purpose Timer
* Centronics Printer Port
* 128K RAM (expandable to 512K on board.)
* Expansion Bus
* 5.75 x 8.0 Inches Mounts to Side of Drive
* +5v 2A, +12 for RS-232
* Power Connector same as disk drive

Add a terminal, disk drive and power, and you will have a powerful 68000 system.

-Software Included:
* K-OS ONE, the 68000 Operating System (source code included)
* Command Processor (w/source)
* Data and File Compatible with MS-DOS
* A 68000 Assembler
* An HTPL Compiler
* A Line Editor

ASSEMBLED AND TESTED ONLY $395.00

* * * * * * * * * * * * * * * * *

K-OS ONE, 68000 OPERATING SYSTEM

For your existing 68000 hardware, you can get the K-OS ONE Operating System package for only $50.00. K-OS ONE is a powerful, pliable, single user operating system with source code provided for operating system and command processor. It allows you to read and write MS-DOS format diskettes with your 68000 system. The package also contains an Assembler, an HTPL (high level language) Compiler, a Line Editor and manual.

SHIPPED ON AN MS-DOS 5 1/4" DISK. $50.00

* * * * * * * * * * * * * * * * *

Order Now:
VISA, MC
(503) 254-2005

HAWTHORNE TECHNOLOGY

8836 S. E. Stark
Portland, Or 97216

TURBO PROGRAMMERS–

Tdebug Plus™...

...CUTS DEBUGGING FRUSTRATION.

*TDebugPLUS is a **new**, interactive symbolic debugger that integrates with Turbo Pascal to let you*

- ***Examine and change variables** at runtime using symbolic names – including records, pointers, arrays, and local variables;*
- ***Trace and set breakpoints** using procedure names or source statements;*
- ***View source code** while debugging;*
- ***Use Turbo Pascal editor and DOS DEBUG commands.***

TDebugPLUS also includes a special MAP file generation mode fully compatible with external debuggers such as Periscope, Atron, Symdeb, and others – even on programs written with Turbo EXTENDER.

An expanded, supported version of the acclaimed public domain program TDEBUG, the TDebugPLUS package includes one DSDD disk, complete source code, a reference card, and an 80-page printed manual. 256K of memory required. Simplify debugging! $60 COMPLETE.

TURBO EXTENDER™

Turbo EXTENDER provides you the following powerful tools to break the 64K barrier:

- ***Large Code Model** allows programs to use all 640K without overlays or chaining, while allowing you to convert existing programs with minimal effort; makes EXE files;*
- ***Make Facility** offers separate compilation eliminating the need for you to recompile unchanged modules;*
- ***Large Data Arrays** automatically manages data arrays up to 30 megabytes as well as any arrays in expanded memory (EMS);*
- ***Additional Turbo EXTENDER tools** include Overlay Analyst, Disk Cache, Pascal Encryptor, Shell File Generator, and File Browser.*

The Turbo EXTENDER package includes two DSDD disks, complete source code, and a 150-page printed manual. Order now! $85 COMPLETE.

TURBOPOWER UTILITIES™

*"If you own Turbo Pascal, you should own TurboPower Programmers Utilities, that's all there is to it." Bruce Webster, **BYTE Magazine***

TurboPower Utilities offers nine powerful programs: Program Structure Analyzer, Execution Timer, Execution Profiler, Pretty Printer, Command Repeater, Pattern Replacer, Difference Finder, File Finder, and Super Directory.

The TurboPower Utilities package includes three DSDD disks, reference card, and manual. $95 with source code; $55 executable only.

ORDER DIRECT TODAY!

- ***MC/VISA Call Toll Free** 7 days a week. 800-538-8157 x830 (US) 800-672-3470 x830 (CA)*
- ***Limited Time Offer!** Buy two or more TurboPower products and save 15%!*
- ***Satisfaction Guaranteed** or your money back within 30 days.*

For Brochures, Dealer or other Information, PO, COD – call or write:

TURBO Power™ *3109 Scotts Valley Dr., #122 Scotts Valley, CA 95066 (408) 438-8608 M-F 9AM-5PM PST*

The above TurboPower products require Turbo Pascal 3.0 (standard, 8087, or BCD) and PC-DOS 2.X or 3.X, and run on the IBM PC/XT/AT and compatibles

The Art of Source Code Generation

Disassembling Z-80 Software

by Clark A. Calkins, C.C. Software

Introduction

What is source code generation all about? Well, I consider this as the process of creating usable program source code in some language that is equivalent to an initial executable program. By usable I mean that the source code has been sufficiently documented or commented such that you or someone else schooled in the language can understand what is going on.

If you already have an executable program, why would anyone want the source code? The truth is, many users would not. But if you ever wanted to change a program, source code makes it much easier. In some cases, it is not practical to change a program without the source. Besides the necessity or desire to change a program, some people would just like to understand how it works.

Why is it that software producers very seldom make source code available for their products? There are several reasons (or excuses) for not releasing the source code. These include 1) "...you don't need it, program XYZ works perfectly as it is", 2) "...you could then make copies and give these to others", 3) "...it's too complicated, you couldn't understand it" and so on. Actually, the real reason is that software writers feel that their programs represent private and creative thoughts and that they loose privacy if the source code gets out of their control. There is a mystique to writing programs and programmers tend to keep it that way. A form of "job security."

However, users do have a legitimate requirement for source code. After all, the programmer wrote the program to do what he/she thought was the best, but the users generally have different ideas and frequently want to change the default values which were selected by the programmer. Some programs come with installation instructions that do allow some flexibility in this regard, but this is not always enough. If you ever wanted to learn how to write an operating system or compiler and only had a book or two for reference, you would then see the advantage of having some source code to refer to. This is indispensable!

Can the source code for any program be produced? Theoretically the answer is yes. However, from a practical standpoint, only a few programs would be worth the effort involved. The idea to keep in mind, is that if the computer can execute the program, it can be disassembled. This is because the process of disassembling a program is really a conversion from one language (machine code) to another. Like translating a book from German to English except there are no ambiguities to worry about. Documenting the program now involves just time and effort (maybe quite a lot of these).

The specific type of source code generation I am going to be dealing with here, is the creation of Z-80 assembly source code. If it were desired to create Pascal or Fortran source code (assuming the program was written in one of these languages), the first step would be to produce assembly instructions and convert groups of instructions into source statements. This would require a thorough understanding of the code produced by the compiler. A difficult task indeed, but it can be done.

In addition, I will be assuming that CP/M is the operating system being used and that it is desired to disassemble a normal transient program and not a ROM or other type of program storage. These are slightly more complicated as the execution address may differ from the physical address.

The disassembler I use is my own (the Masterful Disassembler or MD for short) but most any quality disassembler can be used. It is most handy if the disassembler allows label names to be associated with addresses and also supports comments lines. All disassemblers I know of allow you to define which areas of memory are instructions and which are data (the more data types supported the better). You will find it particularly handy if the disassembler will recognize tables of addresses and put these into its label pool.

If the disassembler you use does not allow label names and comments to be entered, then you will have to work from a printed listing and use an editor to keep track of things. This is a lot more trouble on large disassemblies because up-to-date listings may take hours to generate and you will be tempted to try and get by with older ones. This will sooner or later cause duplication of effort as you re-comment some areas a second time.

For an example, I will be referencing TURBO Pascal® . This is because I am familiar with it and it encompasses many features common to most assembly language programs.

There is a knack to disassembling programs. After you have done one, the next comes easier. Just like programming.

Choosing the Proper Candidates for Dissection

Before setting out to produce source code for that super new program, you first want to know if you can succeed. Before I embark on a new project, I ask myself these questions:

- Is there a real use for the source code for this program?
- Is the program small enough that I can complete this project in a reasonable amount of time?
- Was the program written in assembly or at least compiled with a good (non-threaded) compiler?

Disclaimer

The guidelines contained herein are for educational purposes only. The legality of disassembling a program is not totally free from doubt (although it is done on a routine basis). Software licenses may impose limitiations that the user should be aware of. It is certainly not the intent of this article to deprive any software producers of rightfully earned revenues.

• Is the program static or does it move itself around in memory?

The answer to these questions, when taken together, help me to decide whether or not to try to disassemble a program. I really try to avoid programs that move themselves in memory (like DDT does). I know of no disassembler which can handle these types of programs successfully. The program size is also very important. I have found that, on the average, every processor instruction, uses 2 bytes of memory if it is 8080 code and 1.5 bytes if it is well written Z-80 code (this includes most normal data areas). This means that an 8k program would result in 5333 lines of Z-80 assembly code and when commented, will take up 100k of disk space. I don't recommend users try disassembling programs in excess of 16k (unless they have a VAX to work with!).

How can you tell if a program was originally written in assembly language and is this really important? With some of today's optimizing compilers, this question becomes less important. But in general, when I first look at a disassembly of a program I want to see if there is a consistent flow of logic to the code. When I see code that is composed of a seemingly endless series of calls to subroutines grouped at one end of the program (either low or high addresses), I know this came from a threaded type compiler and will be a bear to disassemble. I like to see subroutine calls followed by code that uses the Z-80 flag register (like "CALL 1234" followed by "JR NZ,4567"). This is the way assembly programmers work but few compilers are smart enough to do this. Most would have inserted a "AND A" or similar instruction. Additionally, compilers produce code that executes under all conceivable conditions. Thus there will be many absolute jump instructions and few relative ones. If you find that you cannot tell if the code was produced by a compiler or an assembler, then it probably does not make much difference.

Over the years, users have suggested various projects for disassembly. These have ranged from fairly reasonable ones like MP/M, OASIS, and MS-DOS to absurd ones like ED and LOTUS-123. Who even uses ED, and who has enough disk space to hold LOTUS?

The Process of Generating Usable Source Code

Prior to disassembling a program, you want to learn as much about how it works as you can. You need to know exactly what the program does and have a reasonable idea of how it does it. When you understand what a program does on a gross level, you will be able to understand the finer details when you get to them.

When I start on a disassembly project, I find it is generally a six step process.

1) Understand what the program does and how it functions.
2) Determine the begin, end, and initial execution addresses.
3) Identify the instruction and data areas.
4) Isolate the subroutines and identify the lowest level ones.
5) Comment and label the subroutines.
6) Comment and label the main program code.

Before you can effectively use a disassembler, you will have to know the bounds of the program you are dealing with. I generally use DDT to get a rough idea of the end address. This will display the address of the physical end of the executable file. The actual end of the program is somewhere before this. Depending on the type of program, it may or may not be easy to locate an effective ending address. At the very least, you can use the physical end address until experience indicates otherwise.

With most disassemblers (especially MD) you will want to identify those areas of the program that contain data and not instructions. You can spot the larger data areas using a hexadecimal and ASCII dump of the program (DDT can be used if the disassembler does not support this). Message areas are quite easy to find, but binary data can be hard if it is not all 00's or 0FFH's. Of special importance is the location of any address tables. When disassembled, these tend to show up as series of LD C,C or other "meaningless" instructions. Depending on the program, the address table might be next to another key table. For example in TURBO Pascal, the editor uses an input command table followed by a corresponding address table. Programmers tend to keep these close together as it makes it easier to write and maintain the program.

An important point to remember is the documentation that came with the program. At times there is a wealth of information here. You will be particularly interested in the following items:

• Error code and error message cross references. What situations lead to the different errors?
• Internal data structure areas. Some of this information may be found in a section which details the interfacing of external programs or subroutines.
• Program options. If there are several options (or "switches") available, what are the defaults and how are they used?
• Names. The manual may use mnemonic names while describing an internal function. For example the TURBO Pascal manual uses FIB while describing file interface blocks. Maintain these same naming conventions.

TURBO Pascal is a good example as the manual goes into considerable detail on the structure of some internal data areas like the file interface blocks and the storage of real and integer type numbers. Plus a memory usage map is provided. Very handy! There are three very gross sections to all programs. These are 1) the initialization section, 2) the main working section, and 3) the wrap-up section.

Initialization

There are certain similarities among most programs when it comes to initialization. One of the first things a program is going to do is setup a machine stack area. This is because CP/M does not define how much stack space is already available. To be safe, programs will generally set their own space aside for this. It is easy to spot. You are going to find a LD SP,nnnn or similar type instruction very early in the initialization phase. Secondly, the memory limits will have to be established. For programs that want to use all available memory, there will something similar to LD HL,(0006) followed by LD (nnnn),HL. This is because CP/M maintains a jump instruction to the BDOS at location 5 and this is the lowest location used by CP/M. Everything below this is available for the program to use. On occasions you will find a program that does not use too much memory and then it merely saves the return address by POP HL and LD (nnnn),HL. This allows the program to return to CP/M without having to do a warm boot. Thirdly, certain data areas will have to be preset, and then the screen will be setup with a sign-on message (this varies the most between programs).

When you disassemble a program, first jump to the initialization area. You will want to identify the stack area which more than likely is close to other data storage areas. You should also be able to locate the screen I/O routines as the program will surely do some kind of screen output during initialization.

There are two common means of outputting a string of data to the screen and you want to find out which method is being used. This will aid in the remainder of the disassembly. The first way is to load the address of the string into a register pair (like HL) and then call a routine which outputs the characters. The other way is to call an output routine and have the string immediately following the call instruction. In either case, the strings will start with a character count byte or terminate with an "unprintable" character (like 00 or 0FFH). CP/M programs commonly use a dollar sign ("$") as a terminator as this is consistent with the BDOS string output function. Determine which method is being used so you can spot and comment these areas as you find them.

You can identify the data areas being initialized but since you probably do not know what these are used for, you cannot add meaningful label names or comments. But have patience, you will be learning a lot more as time goes on.

Main Program Body

This is the heart of any program and the section which is different for every one. Luckily programs are generally written in a modular fashion. Sections of code perform a specific task. This makes it easier for the programmer to write and later debug it. And happily it makes it easier to disassemble. TURBO Pascal had three main portions that were disassembled separately (run-time library, editor, and compiler). By locating the logical divisions of a program, you can minimize the cross communication which hinders the disassembly process.

When you first start on the main program section, keep in mind that a program is a collection of subroutines tied together in a particular order. As you browse through the code initially, try to get an idea of the programming style. Items to keep track of are:

- Are subroutines generally kept in one location (like the beginning or end of the program)?
- Is there a constant pattern of calling subroutines (certain registers always used for arguments)?
- Do routines always terminate in a return instruction? Or do they use jumps to other routines to save time?
- Are certain registers "sacred" and not used by subroutines?
- Do subroutines always preserve registers?

To answer these questions, examine the first few subroutines you find. These are easy to spot. Just check the destination of the call instructions. You do not have to understand the routines right off, but make notes concerning any similarities that stand out. If the program was written in a high level language, there will be many (and often wasteful) similarities.

With these ideas in mind, the next step is to identify as many of the individual subroutines as you can. If you can insert comments with the disassembler, try to separate the routines with a blank comment line or two. Some disassemblers do this automatically when they encounter a return instruction. Most, including MD, do not.

When beginning to disassemble a subroutine and comment it, remember that subroutines are often nested (subroutine A calls subroutine B which calls subroutine C, etc.). You will find it easiest to locate the inner most subroutine (the one that does not call any other subroutine within the program area that has not already been disassembled) and begin there. These are generally smaller with an easier to understand, more direct purpose. You will find that by documenting one of these low level routines and changing the references to it to a meaningful name like "OpenFile", you can go back to a parent subroutine and understand it a little easier. Then go back to its parent subroutine and so on. Because a subroutine is generally referenced by many others, documenting these small ones has a pyramid effect. This bottoms-up process is essential for even the smallest programs.

Initially look for CP/M BDOS references. These are call or jump instructions to location 0005. The process to access CP/M is well defined. The function number is loaded into register C and any parameters use register pair DE. You might have to look a few instructions ahead of the BDOS call (or jump) to find where register C is loaded. Then refer to the CP/M Reference manual for the meaning of this function. Keep a chart on the wall with a summary of these functions. You will refer to it often. By using this approach, you can identify the file routines (open, close, read, write, delete, rename, etc.) and the console routines (character input, character output, status, etc.).

It should not take too long to disassemble and comment the lowest two levels of subroutines. You should then have labeled subroutines like "CreateFile", "PrintChar", "WriteFile", "ConsoleStatus", etc. These will be the work horse of the program as all other subroutines will need to use these.

There will come a time when you seem to be unable to progress with this bottoms-up approach. Maybe by the third or fourth level of subroutines. At this time, you need to call upon your knowledge of what the program does and start a top-down approach. You will need as much information as you can get on how it functions. For example, if you are disassembling an editor (like the editor portion of TURBO Pascal), you know that this will read the keyboard, determine if the key (or keys) constitute a command or data to be added to the file. Then you will look at those subroutines that reference the keyboard (either directly or via a status check subroutine). One of these routines will have to determine if the key is a command key. In TURBO Pascal, commands consist of a control character followed by another character. By examining the data areas initially you have already found the location of the key translation table. Then the subroutine you are looking for will contain a reference to this table. When it is found (it must be there) you then discover that it also references another data area at the same time. Isn't that interesting? Looking at the code shows that as it scans the key translation table, it keeps a count of which one matched (if any). Then when a match is found, it doubles the count and adds it to the start of the second table. Because you know that addresses in the Z-80 are two bytes long (16 bits), you then determine that the second table is a table of subroutine addresses for all possible commands. Now we are really getting somewhere! Looking at the documentation determines what functions are performed for the different commands. All of a sudden you have determined the address and function of many more subroutines. Of course you quickly document and label these.

Every once in a while you may find yourself hopelessly lost within a particular subroutine. At these times, document it as

best you can and put the remainder off until later. As you disassemble the rest of the code, you will find out under what circumstances this subroutine gets executed. This may give you the insight you need to understand it.

Termination

Like the initialization section, the termination portion of a program is fairly standard. The program has to clean up and then get back to CP/M somehow.

The process of cleaning up may be trivial for some programs. Others may have to flush buffers and close data files. Some times the stack pointer is reset to the value it had when the program was first executed. This seems to be common even though it is not required by CP/M. If buffered data output is used, then the program must be sure the buffers are flushed (written to the disk) prior to closing the files. There are two common procedures to flush buffers. The first is to have a separate subroutine that writes the current buffer as it is. This is the "cleanest" method but takes some extra memory. The other process is to write enough end-of-file marks as to be sure to overflow the buffer which causes it to be written. If the buffer holds 1024 bytes, then if that many EOF's are written to the buffer, you "know" that it must have been written. I do not consider this as clean as the first approach as it could result in an extra buffer being written, but it generally takes less memory to implement.

Stumbling Blocks

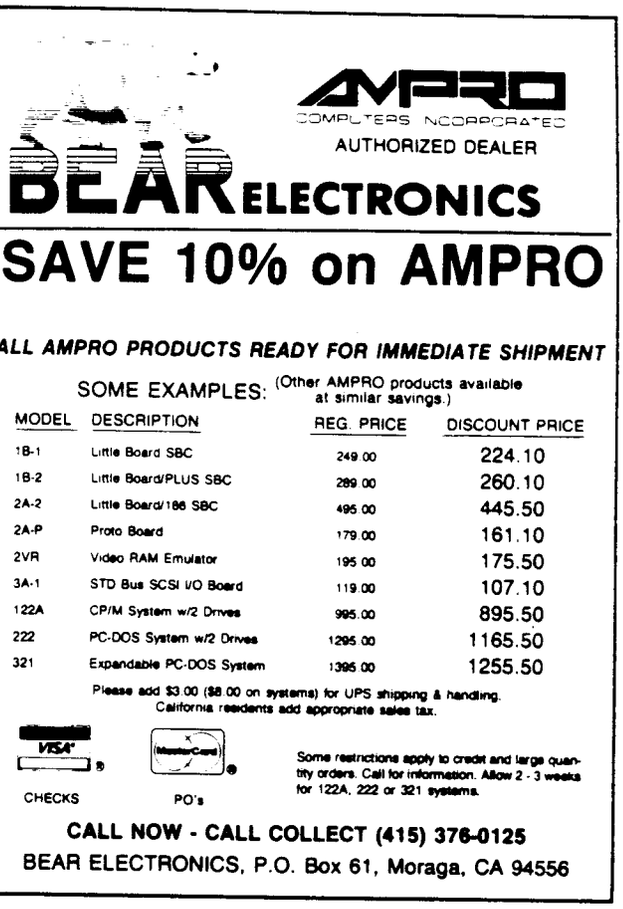
By the very nature of assembly programming, it is possible to write real obscure code. Code can be self-modifying, self-relocating, or a combination of these and you will have fits trying to disassemble it as the code may not function as it first appears.

Self-modifying Code

There are many possibilities here. I will touch on a few of the most common techniques I have run across. Self-modifying code is used to either save memory (possibly increasing execution speed) or to confuse the person trying to disassemble it.

One very common procedure is to alter the contents of a data item being loaded into a register. The instruction "LD A,0" may be changed into "LD A,5" by writing a 5 into the second byte of the instruction. By using this type of code, the execution speed is improved (an immediate load is about twice as fast as an indirect load) and the code is a little smaller. Some disassemblers point out these types of self-modifying code (MD does not). If not, these are easy to spot. Just look for unresolved address references that are within the boundaries of the program code.

A more difficult type of self-modifying code to locate is when the instruction itself is changed. Some subroutines (mainly I/O) have complement functions. For example reading or writing to a file. Because they are very similar, programmers are tempted to "re-use" the same subroutine for both functions. Only a flag or register value indicates which function is to be performed. An example is a file I/O routine that sets a flag based on the contents of register A. The instruction that sets the flag is either an "INC A" or "DEC A". Prior to calling the routine, this instruction is modified to provide the desired function. Since these differ by only a single bit (3C and 3D hex respectively), a clever programmer could have code such as:


AMPRO
COMPUTERS INCORPORATED
AUTHORIZED DEALER
BEAR ELECTRONICS
SAVE 10% on AMPRO
ALL AMPRO PRODUCTS READY FOR IMMEDIATE SHIPMENT
SOME EXAMPLES: (Other AMPRO products available at similar savings.)
MODEL DESCRIPTION REG. PRICE DISCOUNT PRICE
1B-1 Little Board SBC 249.00 224.10
1B-2 Little Board/PLUS SBC 289.00 260.10
2A-2 Little Board/186 SBC 495.00 445.50
2A-P Proto Board 179.00 161.10
2VR Video RAM Emulator 195.00 175.50
3A-1 STD Bus SCSI I/O Board 119.00 107.10
122A CP/M System w/2 Drives 995.00 895.50
222 PC-DOS System w/2 Drives 1295.00 1165.50
321 Expandable PC-DOS System 1395.00 1255.50
Please add $3.00 ($8.00 on systems) for UPS shipping & handling. California residents add appropriate sales tax.
VISA
MasterCard
CHECKS
PO's
Some restrictions apply to credit and large quantity orders. Call for information. Allow 2 - 3 weeks for 122A, 222 or 321 systems.
CALL NOW - CALL COLLECT (415) 376-0125
BEAR ELECTRONICS, P.O. Box 61, Moraga, CA 94556

• Z Best Sellers •

Z80 Turbo Modula-2 (1 disk) $69.95
The best high-level language development system for your Z80-compatible computer. Created by Borland International. High performance, with many advanced features; includes editor, compiler, linker, 552 page manual, and more.

Z-COM (5 disks) $119.00
Easy auto-installation complete Z-System for virtually any Z80 computer presently running CP/M 2.2. In minutes you can be running ZCPR3 and ZRDOS on your machine, enjoying the vast benefits. Includes 80+ utility programs and ZCPR3: The Manual.

Z-Tools (4 disks) $150.00
A bundle of software tools individually priced at $260 total. Includes the ZAS Macro Assembler, ZDM debuggers, REVAS4 disassembler, and ITOZ/ZTOI source code converters. HD64180 support.

PUBLIC ZRDOS (1 disk) $59.50
If you have acquired ZCPR3 for your Z80-compatible system and want to upgrade to full Z-System, all you need is ZRDOS. ZRDOS features elimination of control-C after disk change, public directories, faster execution than CP/M, archive status for easy backup, and more!

DSD (1 disk) $129.95
The premier debugger for your 8080, Z80, or HD64180 systems. Full screen, with windows for RAM, code listing, registers, and stack. We feature ZCPR3 versions of this professional debugger.

Quick Task (3 disks) $249.00
Z80/HD64180 multitasking realtime executive for embedded computer applications. Full source code, no run time fees, site license for development. Comparable to systems from $2000 to $40,000! Request our free Q-T Demonstration Program.

Z-System OEM inquiries invited. Visa/Mastercard accepted. Add $4.00 shipping/handling in North America; actual cost elsewhere. Specify disk format.

Echelon, Inc.
885 N. San Antonio Road • Los Altos, CA 94022
415/948-3820 (Order line and tech support) Telex 4931646

```
LD      HL,FileIO       ;Address of general file i/o.
INC     (HL)            ;Change INC into DEC for reading.
JP      (HL)            ;Execute and return through it.
```

While this type of code would be difficult to manage because it would have to remember the last state, it could be made to function.

Relocatable Code

The Z-80 does not process relocatable code. So the program must handle the relocation task itself before it is executed. While there are many ways to do this, the most common procedure to to maintain a table of those address that have to be relocated. TURBO Pascal, in the overlay command processor (TURBO.OVR) uses this technique. The program is assembled as if it begins at location zero (or some such address). Just prior to being executed, a relocation routine moves it to the destination address (in the case of TURBO.OVR this is high memory) and then changes all address references accordingly. Finally the program is executed.

A second procedure is to maintain a bit map table within the program (some assemblers help you maintain such a table). This table has a bit for every byte in the program. If the bit is set, the corresponding byte must be relocated. This works for programs which are relocated by a certain number of pages (256 bytes to a page). That way only the high byte of an address must be changed.

Trick Code

You may find that assembly programmers love to use certain "tricks" that improve execution speed or save memory. You have to watch out for this or you could get really confused. TURBO Pascal in many places uses instruction sequences like:

```
ReadFile: LD      A,0AFH
          ORG     $-1
WriteFile:XOR     A
          AND     A
          PUSH    AF
```

By jumping into the middle of the instruction at "ReadFile" the program is able to set a flag byte (using register A) that is zero on a write and non-zero on a read. This saves a few bytes and drives the disassembler crazy! Change these into something like:

```
ReadFile: DB      3EH
WriteFile:XOR     A
          AND     A
          PUSH    AF
```

And add some good comments so you will know what is going on the next time you have to look at this.

One way of logging an error message where it is desired to load register A with a value (say from 1 to 5) to correspond to an error code, could be done as follows:

```
Error5:     INC     A
Error4:     INC     A
Error3:     INC     A
Error2:     INC     A
Error1:     INC     A
            LD      (ErrorCode),A
```

To log an error then, the following sequence would work (this saves a single byte for each occurrence).

```
XOR     A
CALL    Error4
```

Initially when you disassemble this type of code sequence, you may be tempted to think that the series of "INC A" instructions are really data. After all it doesn't make much sense at first. But as you uncover those locations where these instructions are referenced, you soon discover what is going on and correct your mistake.

There are plenty of other tricks you may run into and some are even more obscure.

Making Use of the Generated Source Code

Now that you have generated the source code and it has been documented more of less to your liking, what do you do with it? Although you probably should have asked yourself this before you started this project, here is a list of the major uses for the source code to a program.

- Using the source code you can fix those "bugs" that prevented you from fully utilizing the product.
- Set those default parameters to what you want.
- Modify the program to include features that you (and perhaps others) want to see.
- Learn how the programmer did those wonderful things. Make your programs work just as well using these techniques.
- Make some money from your labor.

Because you have converted the program from machine code to an ASCII file suitable for an assembler, you can now edit the file making any changes you wish. Then you reassemble and link (or load) the program. This new copy now functions the way you want it to.

If one of your reasons for producing the source code was to learn how the program functioned, then the printed listing is probably the most important result. While the process of disassembling already gave you great insight into the program as a whole, now you will be able to study that particular aspect that intrigued you (like just how does TURBO Pascal handle a forward label reference when it is only a one pass compiler? Is it really a one pass compiler??).

Another interesting, but often overlooked, use for the source code is that other people may have similar interests to yours. Maybe you could sell it? This is a sticky area because the source code is composed of instructions whose order (ie, the executable part of the program) is owned by the author, plus your added comments and label names. There have been a few examples of disassembled programs being published (for example the Timex/Sinclair ROM) but these most likely required the consent of the original author.

Regarding your work, you would certainly be allowed to sell your label names and comments but probably not the instructions. After all if you sold the instructions then you are, in effect, selling the program and would owe the author royalties. But all is not lost as there is a way out of this mess.

If you consider that every line of the assembly source file looks like this:

```
<-- #1 ---> <---------- #2 -------> <---------- #3 -------> <--- #4--->
(LabelName) (processor instruction) (instruction arguments) (Comments)
```

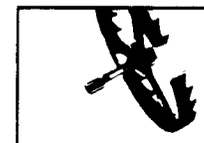
where each field could be absent, then you could extract the portions that you have created (#1 and #4) from the line and eliminate the portions that the program author created (#2 and #3). The extracted portions could be put into data files and sold. Now in order for a customer to use this data (your label names

and comments), he will have to come up with the instruction and argument fields for each line. This means he must already have the program. Since even a rudimentary disassembler has no trouble producing the processor instructions for a program, you will have to come up with a disassembler that knows how to combine your label names and comments with the disassembled instructions. This is not a trivial task, but its not that difficult either. What you come up with is a program that can reconstruct the source code on the customer's computer system.

In this way you would only be distributing your label names and comments along with a simplistic disaassembler that knows how to put the pieces back together again. And, importantly, the author is not cheated out of any royalties that are due because the user must already have a copy of the program. Which hopefully was obtained properly.

The results of this fancy foot work is that the customer has to wait a short while for the source code generator to produce the final files. The generator for TURBO Pascal takes about twenty minutes to produce 21,000 lines of assembly code. Not a big inconvenience considering the immensity of this task.

These knowledgable, highly specialized disassemblers are what I call Source Code Generators. C.C. Software has these available for CP/M 2.2 ($45), CP/M 3 ($75), and TURBO Pascal v 3 (CP/M, Z-80) ($45). And more are in the works. ■



BD Software, Inc., maker of the original

CP/M-80 C Language Development

System, knows

Time is precious

So the compilation, linkage and execution speeds of BDS C are the fastest available, even (especially!) on floppy-based systems. Just ask any user! With 15,000 + packages sold since 1979, there are *lots* of users . . .

New! Ed Ream's RED text editor has been integrated into the package, making BDS C a truly complete, self-contained C development system.

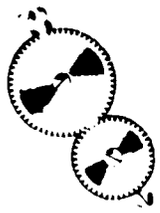
Powerful original features: CDB symbolic source-level debugger, fully customizable library and run-time package (for convenient ROM-ing of code), XMODEM-compatible telecommunications package, and other sample applications.

National C User's Group provides direct access to the wealth of public-domain software written in BDS C, including text editors and formatters, BBS's, assemblers, C compliers, games and much more.

Complete package price: $150.
All soft-sectored disk formats, plus Apple CP/M, available off-the-shelf. Shipping: free, by UPS, within USA for *prepaid* orders. Canada, $5. Other: $25. VISA, MC, COD, rush orders accepted.

BDSoftware, Inc.

BD Software, Inc.
P O Box 2368
Cambridge MA 02238
617 • 576 • 3828



Feedback Control System Analysis

Using Root Locus Analysis and Feedback Loop Compensation

by Bert van den Berg, BV Engineering

Closed loop feedback control systems are used to ensure that an activity occurs as intended. All such systems contain a forward block which performs an action and a feedback block which measures the action and reports the results. The function of the feedback loop is to reduce the error between the measured and desired result to zero or some acceptable value near zero. Figure 1 shows a typical feedback control system in its simplest form.

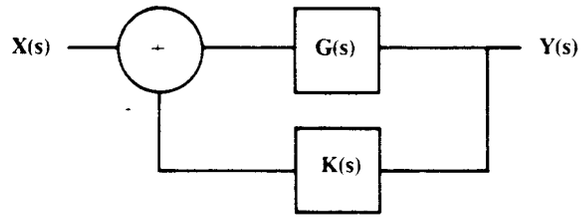


Figure 1 - Simplest Feedback Control Loop

The composite LaPlace transfer function, H(s), of the feedback system of Figure 1 can easily be calculated. If G(s) is the LaPLace transfer function of the forward path and K(s) is the transfer function of the feedback path, and the overall transfer function H(s) is defined to be H(s)=Y(s)/X(s), then:

$$Y(s) = e^{*}G(s) \quad (1)$$

$$e = X(s) - K(s)^{*}Y(s) \quad (2)$$

Substituting (2) into (1) gives:

$$Y(s) = G(s)[\, X(s) - K(s)Y(s) \,] \quad (3)$$

Rearanging (3) gives us the answer for the closed loop transfer function:

$$H(s) = Y(s)/X(s) = G(s)/[\, 1 + K(s)G(s) \,] \quad (4)$$

An example of such a control system is the driver of an automobile. In the case of the driver, he uses the automobile controls to maintain the vehicle on a desired path. The driver's vision provides a means for sensing the actual motion while his brain derives the error between the desired motion and the actual motion. Deviations are corrected, the new path observed, and a new estimate of the error between actual and desired course is provided.

While feedback provides a means to reduce error by measuring the activity, it can also create system instability, cause overshoot, and exhibit undesirable transient response. An example of potential instability and undesirable response would be if the driver of the automobile were drunk or had to rely on information verbally transmitted to him by another observer. With this in mind, it is important to have the capability to assess a control loop's characteristics. In particular, one of the first issues requiring resolution is that of loop stability. Since loop stability, accuracy, and transient response are all related to loop gain, the loop gains for which the loop remains stable are another area of interest. The question of loop stability with respect to loop gain arises because loop element gains may change due to environment and loop accuracy is strongly affected by loop gain.

When the issues related to loop stability have been resolved, the designer then turns to concerns regarding loop transient response. This defines the loop response to a step change in input to the control system. One method for studying and predicting the behavior of closed loop systems is called the root locus method. The root locus method attempts to calculate the roots of the denominator of the closed loop transfer function, H(s), for the control loop. With this method the transfer function is determined in the LaPLace domain (or as a function of the complex variable, s). Next, the roots to the denominator of equation (4) are determined as a function of loop gain. When the roots are plotted in the s plane we can infer some characteristics about stability and transient response.

Roots which lie in the right half of the s plane correspond to an unstable system whose output to a step function input will oscillate with continually increasing amplitude. Those roots lying in the left half plane with complex pairs will exhibit a response to a step input which may overshoot the step amplitude but will settle to the commanded step value. The closer a complex root lies to the imaginary axis the quicker the system response is expected to be. That is, the system will reach its final value quickly. However, the overshoot in reaching the final value may be substantial. The converse is also true. The closer a complex root lies to the real axis, the slower its step response becomes. The type of response for a system having roots located at various points in the s plane is shown graphically in Figure 2.

We will now analyze a typical control loop, compute the root locus of the system using the LOCIPRO(1) root locus program. We will find that the loop is unstable. This behavior will have been predicted from the system root locus by LOCIPRO. A lead-lag loop compensator is then added and the new locus of roots for the compensated system computed. The new root locus provides us with information as to the optimal loop gain and

(1) LOCIPRO, SPP, and PDP, are Professional Engineering Software products from BV Engineering, 2200 Business Way, Suite 207, Riverside, CA 92501, phone (714) 781-0252

relative stability of the compensated system. SPP will be used to verify that the response of the system is now acceptable. PDP is used to make all the plots for these examples.

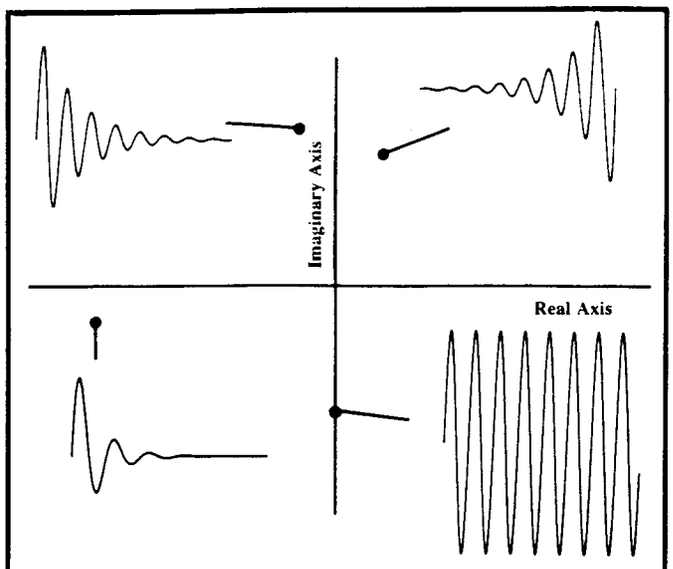


Figure 2 - Typical System Responses

The stable platform control system shown in Figure 3 consists of a platform, an angular rate sensor, and a torque motor to move the platform. In this system a desired rate is input to the system, and the amplifiers feed a signal to the torque motor which in turn causes the platform to rotate at a particular angular rate. The actual rate is fed back to the control loop to allow the loop to accurately follow the commanded rate.

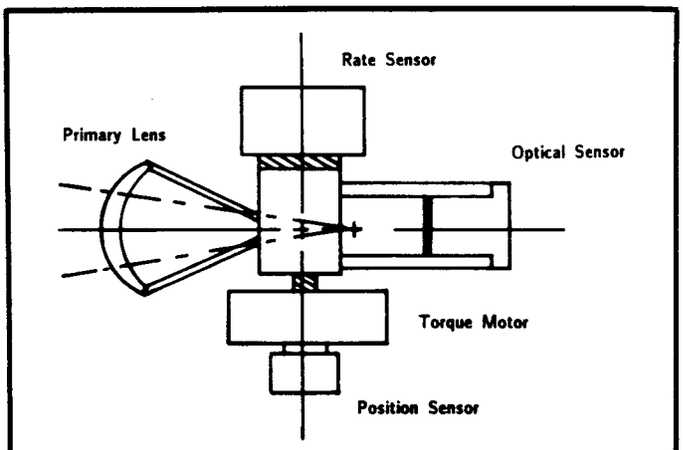


Figure 3 - Stable Platform Control Feedback System

It is desirable to have as high a loop gain as possible in order to maximize steady state accuracy of the closed loop. The system requirements demand that the system settle to the desired steady state final value within 0.1 seconds for a step input. Figure 4 shows the block diagram of the components of the loop in the LaPlace domain. The loop elements describe the characteristics of the gain stage, power amplifier, the torque motor, platform inertia, rate sensor, etc.

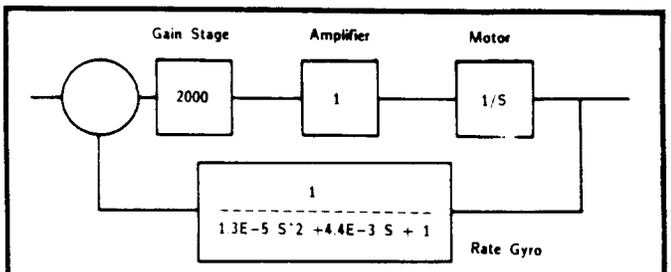


Figure 4 - Loop Block Diagram

The transfer functions describing the loop elements of Figure 4 are entered into the LOCIPRO program and the program is used to compute the root locus as a function of gain. The value of the gain, K, (block 1 of figure 4) is allowed to vary between the limits of 500 and 4000.

After data for the last transfer function block has been entered into LOCIPRO, the program lists all the data and requests whether there is any need to edit the data. Once editing (if any) has been completed, LOCIPRO will list the composite open loop transfer function. The closed loop transfer functions may be saved to data files for future use by LOCIPRO or the Signal Processing Program (SPP). After the root locus has been computed, the pole-zero vs. gain data may be vectored to the display, the printer, or to an ASCII data file. The ASCII data files may be plotted using PDP, PCPLOT, PLOTPRO, or other plotting routines.

The composite open loop transfer function for the system described by figure 4 is computed by LOCIPRO and is listed here:

Open Loop Transfer Function Data

Numerator Coefficients - Ascending Powers of S

Exponent of S	Coefficients
0	.100000E 1

Denominator Coefficients - Ascending Powers of S

Exponent of S	Coefficients
0	.000000E 0
1	.100000E 1
2	.440000E-2
3	.130000E-4

We are now ready to perform an assesment of the stability of the loop with respect to changing gains and have a choice of three levels of resolution in choosing gain steps. The LOCIPRO program can be directed to choose low or high resolution steps with the gain steps automatically selected by the program. The user can also specify the number of steps manually. For this example LOCIPRO was directed to compute a root locus for 70 gain steps for the stable platform problem. The gain K, was

specified to vary between 500 and 4000. The results are shown plotted in the s plane of figure 5.

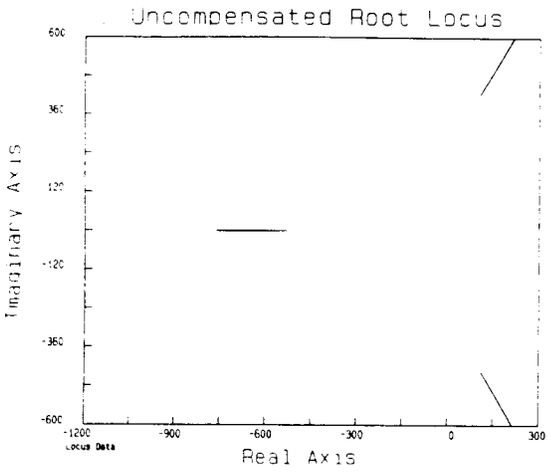


Figure 5 - Root Locus for Uncompensated Stable Platform

A number of different things can be seen from the data presented in figure 5. It is immediately obvious that the system is unstable for all values of gain in the range of 500 to 4000. There are two poles in the right-half plane for all values of gain. These right-half plane poles start at about 27 ± 300i for a gain of 500 and migrate to 200 ± 600i for a gain of 4000. There is also a pole which starts at −400 +0i at a gain of 500 and migrates to −750 +0i at a gain of 4000. Since this third pole lies entirely in the left-half plane it does not worry us as far as stability is concerned.

This system might be stable for loop gains below 500 but we would lose the benefit of accuracy at lower loop gains. A gain of about 2000 would be desirable and thus it will be necessary to "compensate" the loop to improve the stability margin. The design of loop compensators in general is a topic worthy of an article in itself. We cannot do such a topic justice in a few words, and we will not try to do so here. Let it suffice that many loop compensators are just "lead-lag" networks which improve phase margin for the loop by delaying the point at which the poles move into the right-half plane.

A lead-lag compensator for the loop described by Figure 4 was designed to stabilize the loop. This loop compensator having a transfer function C(s)=[0.1S + 1]/S is added to the loop giving us the block diagram shown in Figure 6.

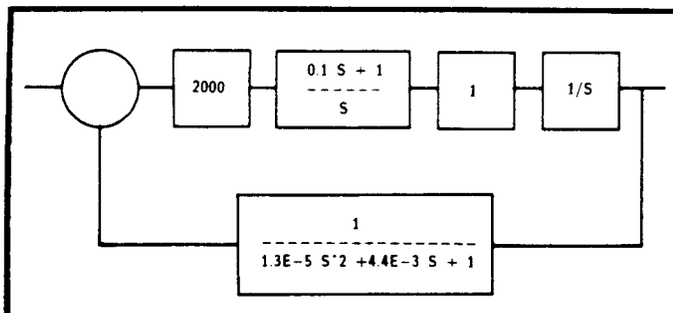


Figure 6 - Compensated Loop Block Diagram

The data for the feedback system described by the transfer functions of Figure 6 are now entered into the LOCIPRO program and plotted using the Plotter Driver Program (PDP). The results are shown in Figure 7.

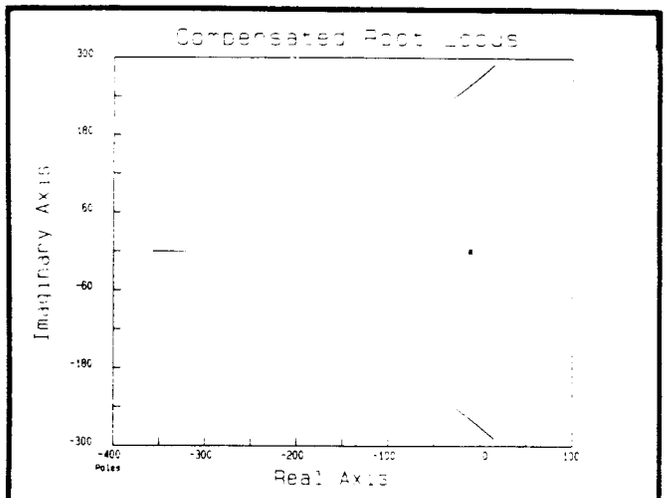


Figure 7 - Root Locus of Compensated System

Note that the plot of Figure 7 is made on a different scale as that of Figure 5. The loop compensator has "pulled" the poles closer to the origin. The additional "zero" at −10 +0i results in a drastically different starting point for the closed loop poles. As the gain K is increased, the poles migrate over a different path as before and stay much closer to the origin.

The plot of Figure 7 shows that the compensated loop is stable for loop gain of 500 and unstable for a loop gain of 4000. Printing the LOCIPRO data in a tabular format, it is easy to see that a complex pair of poles migrate into the right-half plane at a loop gain of 3286. At a loop gain of 2000 the poles are located at −33 ± 235i. Therefore the loop should be stable at a gain of 2000. This much gain should provide the desired accuracy. Having the roots near the imaginary axis predicts a fast response. We only have to see if the transient response meets the 0.1 second settling time requirement.

We use the LOCIPRO program to save the closed loop transfer function for the compensated loop and then use the Signal Processing Program (SPP) to compute the step response. The time domain waveform we wish to compute the response for is generated within SPP (or some other program). Although any user-defined waveshape could have been used, in this case a rectangular waveform was selected. The closed loop transfer function file created by LOCIPRO is read into SPP and the time domain response computed. SPP performs this operation in 35 seconds by using Fast Fourier Transform (FFT) techniques.

SPP takes the rectangular wave and performs a 512 point FFT on the signal to determine the constituent frequencies, and the magnitude and phase of each of these frequencies. The resulting frequency components are called the signals "spectra." The resulting complex spectra is then "filtered" through the transfer function one frequency at a time. The transfer function modifies each frequency component of the original time domain waveform. The resulting frequency spectra is then converted back into the time domain by performing an inverse FFT. The spectra multiplication technique in the frequency domain is identical to convolution in the time domain, where the

rectangular input signal is "convolved" with the impulse response of the closed loop transfer function. Spectra multiplication is far faster than convolution and it takes SPP only 35 seconds to perform all three operations (FFT-Filter-IFFT). The results of the transient analysis of the compensated system is shown in Figure 8.

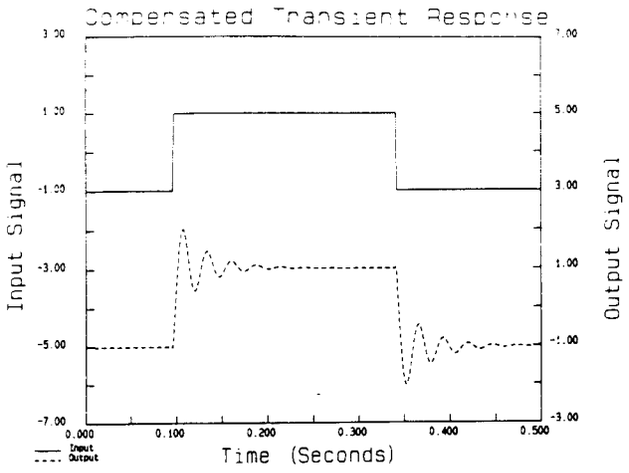


Figure 8 - Transient Response of Stable Platform

The platform output closely tracks the desired input and settles to a steady-state rate within the required 0.1 seconds. With a loop gain of 2000 this "tight'"loop should track the commanded rate quit closely. The gain margin for the loop is 20*Log (3286/2000) = 4.3 dB, not high — but acceptable under most conditions.

This concludes the discussion of root locus techniques and the example using the LOCIPRO program. Several books which are good reading should you desire a more in-depth knowlege of root locus techniques are listed below:

References:

Dorf, R.C., (1967), Modern Control Systems (Addison Wesley)Cannon, R.H. Jr. (1967), Dynamics of Pkysical Systems (McGraw-Hill)Melsa, J.L., and Jones, S.K., (1973) Computer Programs for Computational Assistance in the Study of Linear Control Theory (McGraw-Hill) ■

A book with

ADVANCED TOPICS in TURBO PASCAL

Turbo Pascal - Advanced Applications

Table of Contents

- Optimization Techniques
- Using the DOS Background Print Spooler
- System Level Tools
- Creating Libraries
- Exploiting Command Line Arguments
- Using a Binary Search Tree
- Techniques for Data Compression
- Claiming CP/M Memory
- Break the 64K Data Limit
- Linked Lists for Data Structuring
- Interrupts from Turbo Pascal
- Calling the DOS Command Processor
- Bit Mapped Graphics
- Teaching an Old Screen New Tricks
- Implementing 2D Core Graphics
- Build a Subset Pascal Compiler

Order ***Turbo Pascal - Advanced Applications*** for $19.95 postpaid in USA; with MS DOS disk, $32.95. Add $3.50 for surface shipment to Canada or other countries; air rates on request. Order from **Rockland Publishing, Inc.**, 190 Sullivan, Suite 103, Columbia Falls, MT. 59912. Visa or Mastercard accepted. Phone orders , call (406) 257-9119. **Free information is available.** Dealer inquiries welcomed.

"...packed with good advanced technical information."

NOT ANOTHER BEGINNER TUTORIAL

Turbo Pascal - Advanced Applications

The C Column

A Graphics Primitive Package

by Donald Howes

It's now December 18, and I'm frantically typing this column so I can leave on the 20th to go to Canada for Christmas. By the time you read this, the holiday season will have passed. I hope that it has been a happy one for all. Hopefully, I won't still be chained to this desk.

In this column, I'll start to present code for a graphics primitive package for the IBM PC. While these particular primitives are for that machine, it shouldn't be hard to adapt the code to run on other machines, either MS-DOS based or other operating systems. Once the primitives are complete (which will be this and the next column) we can take a stab at developing some applications using them.

By hiding the machine dependent code at the lowest level, it will free us to develop the important things in a more general and machine independent fashion, aiding in the porting of our software from one system to another. This time, I'll present code for the most basic parts of the primitive package. These routines initialize and terminate the application, provide move and draw routines, and the ability to activate a pixel and clip a line (see Listings 1 and 2).

Getting Things Started (and Stopped)

To start things off, it is necessary to initialize the graphics application. The function graf_init() does this, setting the current x and y location, the maximum and minimum x and y coordinate values, getting the display mode parameters operative when the graphics application started and then setting the system into high resolution graphics mode.

I must admit, I've cheated a bit on this function, since I've hard-wired the initialization routine for an Enhanced Graphics Adapter (EGA). For those not using an IBM or compatible system, this card provides higher resolution and more colors in both the foreground and background than the Color Graphics Adapter (a palette of 16/64). It would be possible to write an initialization routine that could check whether the system had a CGA or EGA video card, but there are differences between the cards that go beyond resolution that would complicate some of the other routines (I also don't have a CGA card to test the code I would write).

The int86() function accesses the DOS interrupts and is used here to access interrupt 10H (ROM BIOS video interrupt). Users of non-IBM systems will have to find the appropriate means to do the analogous on their machines. The first call finds out what the video mode and video page are when the application starts, so that the same environment can be restored on exit. The second call sets the video card into the EGA high resolution 640×350 mode.

Once the application is complete, the function graf_term() restores the original video mode and video page using the information discovered in graf_init(). Resetting the video mode will automatically set the page to 0, so, if a video page other than the first was being used, it has to be set independently.

Where Do We Go From Here?

Now that we know how to start and stop, what do we do while we're there? Let's start with the function for setting a pixel, do_pixel(). This particular routine is very straight forward, being a single call to int86(). The parameters passed are the x,y coordinate of the pixel and the color you wish it to be (#defines for valid colors are given in graph.h, Listing 1).

Equally simple is the move() function, which updates the value of cur_x and cur_y to the x and y coordinates passed to it. Note that both the move() function and the draw() function I'll be talking about in a minute are absolute routines. That is, the x,y value passed is a screen coordinate, not an offset. These functions can be augmented by relative ones (which do use an offset) very easily. Using the move() function as an example, it would be:

```
void move_rel(xoff,yoff)
int xoff, yoff;
{
    int x, y;
    x = cur_x + (xoff);
    y = cur_y + (yoff);
    move(x,y);
}
```

A function draw_rel() would be identical, except calling draw(), instead of move().

I'll discuss draw() and clip() at the same time, since clip() is used by draw() to clip lines to the screen. The draw() function uses the Bresenham line algorithm to draw a line from the current x,y location to that specified. This algorithm is quick, since it uses only integer arithmetic to calculate which pixels are part of the line and avoids the "stair step" effect which plagues other integer based line drawing routines. An extended discussion of this routine can be found in Foley and Van Dam, pg. 433-436 (see References section below).

The clip() function is used to clip a line segment to a window, so that no time is wasted in calculating pixel positions which fall outside the window and are not displayed. The function uses the Cohen-Sutherland clipping algorithm (see Newman and Sproull, pg. 65-67) to calculate the length of the line which is visible within the window (in this case, the window is the same as the physical screen, the viewport). This clipping is an iterative process, where the line is successively truncated until both endpoints of the line are within the window. The function code() returns the 4-byte values which are used to determine when the line is within the window. In addition, the clip() function tests for the situation where line clipping results in a line where both end points are on the same side of the window and are external to it. In this case, the line cannot be displayed and the function returns.

Listing 1: Header File For Graphics Primitives

```
/* header file for graphic primitives */

#include                <stdlib.h>
#include                <stdio.h>
#include                <dos.h>

int cur_x, cur_y;                       /* current x, y coordinates */
int x_max, x_min, y_max, y_min;    /* max and min screen values */
int page, mode;                         /* video page and mode at startup */

static union REGS inreg, outreg;        /* unions for int86() */

/* defines for foreground colors */

#define         BLACK           0
#define         BLUE            1
#define         GREEN           2
#define         CYAN            3
#define         RED             4
#define         MAGENTA         5
#define         BROWN           6
#define         WHITE           7
#define         GREY            8
#define         LBLUE           9
#define         LGREEN          10
#define         LCYAN           11
#define         PINK            12
#define         LMAGENTA        13
#define         YELLOW          14
#define         BWHITE          15

void graf_init(), graf_term(), do_pixel(), move(), draw();
int clip(), code();
```

Listing 2: Graphics Primitives.

```
/*****************************************************************************
*                                                                            *
*       graph.c :       graphics primitives for the IBM PC.                  *
*                                                                            *
*       Copyright (C) 1986      Donald Howes                                 *
*               All Rights Reserved.                                         *
*                                                                            *
*       This program may be copied for personal, non-commercial use only,    *
*       provided that the copyright notice is included in all copies.        *
*                                                                            *
*       Note : most of these functions use IBM PC Video ROM BIOS calls to    *
*          operate and are specific to the IBM PC and close compatibles.     *
*                                                                            *
*****************************************************************************/

#include                "graph.h"

/*  graf_init   - initializes the screen

   description - sets the screen to 640x350 mode and initializes variables.
                 Also saves previous parameters for restoration.

C Column 27 Listing 2

*/
void graf_init()
{
        cur_x = cur_y = 0;              /* set current x,y location */

        x_min = y_min = 0;              /* minimum values for x and y */
        y_max = 349;                    /* maximum y value */
        x_max = 639;                    /* maximum x value */

        inreg.h.ah = 0x0f;              /* get current display mode */
        int86(0x10,&inreg,&outreg);     /* ROM video call 10H */
        page = outreg.h.bh;             /* monitor page to reset to */
        mode = outreg.h.al;             /* monitor mode to reset to */

        inreg.h.ah = 0;
        inreg.h.al = 0x10;              /* set EGA high res mode */
        int86(0x10,&inreg,&outreg);     /* 640 x 350 */
}       /* end of graf_init() */


/*      graf_term -             reset the system.

        description -   reset the system to the way it was before the
                                        initialization call.
*/

vvoid graf_term()
{
        inreg.h.ah = 0;                         /* reset the mode */
        inreg.h.al = mode;
        int86(0x10,&inreg,&outreg);

        if (page)               /* if page wasn't the first (0), reset */
        {
                inreg.h.ah = 5;         /* select display page */
                inreg.h.al = page;
                int86(0x10,&inreg,&outreg);
        }
}       /* end of graf_term() */

/*      do_pixel                - does a pixel

        description             - turns the given pixel to the specified color.
*/

void do_pixel(x,y,color)
int x, y, color;
{
        /* set the new color using int86() */

        inreg.h.ah = 0xc;
        inreg.h.al = color;
        inreg.x.cx = x;
        inreg.x.dx = y;
        int86(0x10,&inreg,&outreg);
}       /* end of do_pixel() */
```

C Column 27 Listing 2

```
/*      code - used by Cohen/Sutherland clipping algorithm.

        description - provides the four bit code needed to determine where
                      the line segment falls in relation to a viewport.
*/

int code(x,y)
int x, y;
{
        return(x<x_min)<<3 | (x>x_max)<<2 | (y<y_min)<<1 | (y>y_max);
}       /* end of code() */

/*      clip - clips a line.

        description - clips a line to a viewport using the Cohen/Sutherland
                      algorithm.
*/

int clip(x1,y1,x2,y2)
int x1, y1, x2, y2;
{
        void move();
        int code();
        int c1 = code(x1,y1);
        int c2 = code(x2,y2);
        int dx, dy;

        while (c1 | c2)
        {
                if (c1 & c2)
                        return;
                dx = x2 - x1;
                dy = y2 - y1;
                if (c1)
                {
                        if (x1 < x_min)
                        {
                                y1 += dy * (x_min - x1) / dx;
                                x1 = x_min;
                        }
                        else if (x1 > x_max)
                        {
                                y1 += dy * (x_max - x1) / dx;
                                x1 = x_max;
                        }
                        else if (y1 < y_min)
                        {
                                x1 += dx * (y_min - y1) / dy;
                                y1 = y_min;
                        }
                        else if (y1 > y_max)
                        {
                                x1 += dx * (y_max - y1) / dy;
                                y1 = y_max;
                        }
                }
                else
                {
                        if (x2 < x_min)
                        {
                                y2 += dy * (x_min - x2) / dx;
                                x2 = x_min;
                        }
                        else if (x2 > x_max)
                        {
                                y2 += dy * (x_max - x2) / dx;
                                x2 = x_max;
                        }
                        else if (y2 < y_min)
                        {
                                x2 += dx * (y_min - y2) / dy;
                                y2 = y_min;
                        }
                        else if (y2 > y_max)
                        {
                                x2 += dx * (y_max - y2) / dy;
                                y2 = y_max;
                        }
                }
        }
        move(x1,y1);
}       /* end of clip() */

/*  moove - performs move
        description - performs a move (does not draw) and resets the values
                      for cur_x and cur_y.
*/

void move(x,y)
int x, y;
{
        cur_x = x;
        cur_y = y;
}       /* end of move() */


/*      draw - draws a line using the Bresenham algorithm.

        description - draws a line from the current location to the location
                      referenced by (x2,y2). It also updates cur_x and cur_y.

*/

void draw(x2,y2,color)
int x2,y2,color;
{
        void do_pixel(), move();
    int dx, dy, incr1, incr2, incr3, d, x, x1, y, y1, xend, yend;

        clip(cur_x,cur_y,x2,y2);

        x1 = cur_x;
        y1 = cur_y;
    dx = abs(x2-x1);
    dy = abs(y2-y1);
```

C Column 27 Listing 2

```
  if (dy <= dx)
  {                     /* absolute value of slope is < 1 */
      if (x1 > x2)      /* start at point with smaller x co-ord. */
      {
          x = x2;
          y = y2;
          xend = x1;
          dy = y2 - y1;
      }
      else
      {
          x = x1;
          y = y1;
          xend = x2;
          dy = y2 - y1;
      }
      d = (dy >= 0) ? incr1 - dx : incr1 + dx;
      incr1 = dy << 1;
      incr2 = (dy-dx) << 1;
      incr3 = (dy+dx) << 1;

      do_pixel(x,y,color);

      while (x < xend)
      {
          x ++;
          if (d >= 0)
          {
              if (dy <= 0)          /* negative or zero slope */
                 d += incr1;
              else                  /* positive slope */
              {
                  y ++;
                  d += incr2;
              }
          }
          else
          {
              if (dy >= 0)
                 d += incr1;
              else
              {
                  y --;
                  d += incr3;
              }
          }
          do_pixel(x,y,color);
      }
  }
  else
  {
      if (y1 > y2)
      {
          y = y2;
          x = x2;
          yend = y1;
          dx = x1 - x2;
      }
      else
      {
          y = y1;
          x = x1;
          yend = y2;
                  dx = x2 - x1;
      }
      d = (dx >= 0) ? incr1 - dy : incr1 + dy;
      incr1 = dx << 1;
      incr2 = (dx-dy) << 1;
      incr3 = (dx+dy) << 1;
      do_pixel(x,y,color);
      while (y < yend)
      {
          y ++;
          if (d >= 0)
          {
              if (dx <= 0)
                 d += incr1;
              else
              {
                  x ++;
                  d += incr2;
              }
          }
          else
          {
              if (dx >= 0)
                 d += incr1;
              else
              {
                  x --;
                  d += incr3;
              }
          }
          do_pixel(x,y,color);
      }
  }
      move(x2,y2);
  }   /* end of draw() */
```

(C Column continued)

Graphics References

This has been a rapid introduction to graphics. For those who wish to delve a little deeper into the subject or want to try their hand at developing their own routines, here are some references you will find helpful.

Bruce A Artwick, 1984, "Applied Concepts in Microcomputer Graphics", Prentice-Hall, Inc., Englewood Cliffs.

J.D. Foley and A. Van Dam, 1982, "Fundamentals of Interactive Computer Graphics", Addison-Wesley Publishing Company, Reading.

Steve Harrington, 1983, "Computer Graphics : A Programming Approach", McGraw-Hill Book Company, New York.

William M. Newman and Robert F. Sproull, 1979, "Principles of Interactive Computer Graphics", McGraw-Hill Book Company, New York.

That's it for this time. The next column will finish up the primitive package and we can move on to some interesting applications. ■

Surplus Parts Resource

Here's a catalog any serious computer tinkerer needs. It's a treasure-trove of stepper motors, gear motors, bearings, gears, power supplies, lab items, parts and pieces of mechanical and electrical assemblies, science doo-dads, goofy things, plus project boxes, lamps, lights, switches, computer furniture, and stuff you might have never realized you needed.

All at deep discounts cause they are surplus!

Published every couple of months, and consecutive issues are completely different. Send $1.00 for next three issues.

JERRYCO, INC. 601 Linden Place, Evanston, Illinois 60202

EVERYTHING YOU NEED . . . $279.00

Now it's easy to program the Heath-Zenith HERO-1® Robot with an Apple® II. HERO® Macros for the S-C Software 6800 Cross Assembler program in Heath's Robot Interpreter Language with easily remembered mnemonics. The HERO® Macros come with 30 pages of documentation.

[For example, the line 1130 – MVWRIM GRIP OPEN, 60, FAST instructs the HERO® to open his gripper 60 units at fast speed. Motor position is expressed in base-10.]

Transfer to HERO® with ROBI . . . an affordable interface for the robotics experimenter . . . is simple.

- **ROBI is a complete package. No additional hardware required for Apple® or HERO®.**
- **ROBI installs quickly in an Apple® II, II+, or IIe. Once installed, no hardware changes are needed. Within minutes, you will be programming HERO®.**
- **With ROBI and the Cross Assembler, the programmer uses Apple®'s memory to write the program, and HERO®'s memory to run the program.**
- **Not "copy protected," archival copies may be made as needed.**
- **ROBI offers expansion potential.**

VISA and MasterCard accepted.

BERSEARCH Information Services

26160 Edelweiss Circle
Evergreen, Colorado 80439

The Cross Assembler with HERO® Macros sells for $100.00; the ROBI Interface sells for $199.00. Both as a package – $279.00.

To order, or for more information, call (303) 670-6137.

APPLE® is a trademark of Apple Computer. HERO® is a trademark of [illegible]

The Hitachi HD64180

New Life for 8-bit Systems

by Jon Schneider

This article was written with the intent of giving those unfamiliar with Hitachi's new 8 bit microprocessor an understanding of its advantages over the Zilog Z-80. Also, through the use of some actual 64180 bios level code, an understanding of how to utilize some of its advanced features.

The Hitachi 64180 is a highly integrated 8 bit CMOS based micro-processor designed to be fully compatible with the large base of Z-80 software already in existence. In addition to an enhanced instruction set and increased throughput, the chip performs many of the functions that previously required a large number of support chips. A few of those functions are:

- On chip MMU, supporting up to 512 bytes of linear address space
- Two channel DMA
- Programmable RAM refresh rates
- Programmable memory and I/O wait states
- Two fully programmable 16 bit reload timers
- Two full duplex asynchronous serial ports
- One clocked serial I/O port
- Interrupt controller (can use reload timers to generate interrupts)
- Ability to relocate on-chip I/O addresses

Many of the features of the chip serve to make the design and production of a system easier and cheaper for the manufacturer, and will be transparent to the applications level programmer. However, if you are a systems level programmer, or desire to use any of the memory above 64k in any applications, then you will need to become familiar with the methods used to control the various on-chip functions.

In addition to the on-chip ports used to control the additional integrated hardware, there is an extended instruction set. Many of the new instructions are directly related to on-chip port I/O, and will be of no real use other than to avoid I/O address conflicts between the on-chip and external ports, or to make sure that the high order bits of the address bus are all 0 (if ANYTHING is on the high 8 bits of the address bus, the Hitachi's ports won't be accessed).

There are several new instructions that will be of use to any assembly level programmer. They are:

- MLT — multiply two 8 bit values, with 16 bit result
- TSTIO — AND the contents of a specified I/O port with the accumulator
- TST (g) — AND specified register and accumulator
- TST (m) — AND immediate data and accumulator
- TST (HL) — AND memory and accumulator

All of the TST instructions are non-destructive on the accumulator, eliminating the need to save and/or reload the accumulator after the AND operation. The flags are set-reset in the PSW as would be expected for any other AND operation. MLT will be covered in more detail later in this article.

The 64180 uses 64 I/O addresses to control its internal ports, and allows relocation of those addresses to any 64 byte boundary within the first 256 bytes of the 64k I/O address space. This was done to facilitate the add-on of the Hitachi processor to existing hardware without causing I/O address conflicts.

Due to the large number of internal ports, I use an include file in all of my 64180 work, and refer to the ports by their names as shown in the Hitachi manual. This file is shown in Listing 1, and assumes that the ports start at address 0.

The addition of a few new instructions would hardly qualify the Hitachi as a marked improvement over the venerable Z80, and other than than making the design of a new system easier, it wouldn't be generating the interest that it is without something to make it much more powerful than the current 8 bit workhorse.

That feature is its ability to address up to 512k bytes of memory in a linear fashion. Due to the lack of more than 16 bits of addressing for the programmer, you are still limited to 64k of memory addressable at one time, but developers can now code programs to utilize memory past 64k in a uniform manner, much like PC-DOS programmers now do.

If you are not familiar with how the IBM PC and other 8088 based computers address memory, you may be surprised to discover that they have been working within the same limitation of a 64k segmented architecture. The main difference is that the 8088 based systems don't actually have to switch memory in and out of one 64k segment, but have to use a segment register in addition to a 16 bit address to specify an absolute address.

You may have noticed that I mentioned 512k bytes of LINEAR address space for the Hitachi, and then turned right around and said that the memory above 64k had to be switched in and out of the 64k address space the programmer uses. That is not as contradictory as it may seem.

The address bus on the Hitachi is actually 19 bits wide, allowing the processor direct linear access to any address within 512k bytes. It is the programmer that has to work within the limits of addressing memory with only 16 bits. Even that isn't entirely accurate, as some operations, such as DMA, allow the programmer to specify the absolute addresses within the 512k byte address space.

Since addressing the 64180's extra memory is one of the more confusing aspects of programming the chip, I will cover it in detail. The manual as supplied by Hitachi is somewhat confusing and terse, to say the least.

The 64k of memory that is accessible at any one time is divided into three separate areas called Common Area 0, Common Area 1, and the Bank Area. How 512k bytes of physical memory is mapped into a logical 64K of address space is deter-

FOR TRS-80 MODELS 1, 3, 4, 4P
IBM PC/XT, AT&T 6300, ETC.

The MMSFORTH System. Compare.

- A total software environment: custom drivers for printer, video and keyboard improve speed and flexibility. (New TRS-80 M.4 version, too!)
- Common SYS format gives you a big 395K (195K single-sided) per disk, plus a boot track!
- Common wordset (79-Standard plus MMSFORTH extensions) on all supported computers.
- Common and powerful applications programs available (most with MMSFORTH source code) so you can use them compatibly (with the same data disks) across all supported computers.
- Very fast compile speeds and advanced program development environment.
- A fantastic full-screen Forth Editor: Auto-Find (or -Replace) any word (forward or back), compare or Pairs-Edit any two ranges of blocks, much more.
- Temporary dictionary areas.
- QUANs, VECTs, vectored I/O, and many more of the latest high-performance Forth constructs.
- Manual and demo programs are bigger and better than ever!
- Same thorough support: Users Newsletter, User Groups worldwide, telephone tips. Full consulting services.
- Personal Licensing (one person on one computer) is standard. Corporate Site Licensing and Bulk Distribution Licensing available to professional users.

MMSFORTH

A World of Difference!

The total software environment for IBM PC/XT, TRS-80 Model 1, 3, 4 and close friends.

- Personal License (required):
  - MMSFORTH V2.4 System Disk $179.95
  (TRS-80 Model 1 requires lowercase, DDEN, 1.40-track drive.)
- Personal License (additional modules):
  - FORTHCOM communications module . . . . $ 49.95
  - UTILITIES 49.95
  - GAMES 39.95
  - EXPERT-2 expert system 69.95
  - DATAHANDLER 99.95
  - DATAHANDLER-PLUS (PC only, 128K req.) 99.95
  - FORTHWRITE word processor 99.95
- Corporate Site License Extensions . . . . . from $1,000
- Bulk Distribution . . . from $999/50 units.
- Some recommended Forth books:
  - FORTH: A TEXT & REF. (best text!) . . . . $ 18.95
  - THINKING FORTH (best on technique) . . . 16.95
  - STARTING FORTH (popular text) 19.95

Shipping/handling & tax extra. No returns on software.

Ask your dealer to show you the world of MMSFORTH, or request our free brochure.

MILLER MICROCOMPUTER SERVICES
61 Lake Shore Road, Natick, MA 01760
(617) 653-6136

Listing 1

```
; HD64180 Ports
; -------------
; Asynchronous Serial Communication Interface (ASCI)
; --------------------------------------------------
cntla0  equ     0       ; ASCI Control Reg A Chnl 0
cntla1  equ     1       ; ASCI Control Reg A Chnl 1
cntlb0  equ     2       ; ASCI Control Reg B Chnl 0
cntlb1  equ     3       ; ASCI Control Reg B Chnl 1
stat0   equ     4       ; ASCI Status Reg Chnl 0
stat1   equ     5       ; ASCI Status Reg Chnl 1
tdr0    equ     6       ; ASCI Transmit Data Reg Chnl 0
tdr1    equ     7       ; ASCI Transmit Data Reg Chnl 1
rdr0    equ     8       ; ASCI Receive Data Reg Chnl 0
rdr1    equ     9       ; ASCI Receive Data Reg Chnl 1

; Clocked Serial Input/Output Port (CSI/O)
; ----------------------------------------
cntr    equ     0ah     ; CSI/O Control Reg
trdr    equ     0bh     ; CSI/O Transmit/Receive Data Reg

; Programmable Reload Timer (PRT)
; -------------------------------
tmdr0l  equ     0ch     ; PRT Timer Data Reg Chnl 0 Low
tmdr0h  equ     0dh     ; PRT Timer Data Reg Chnl 0 High
rldr0l  equ     0eh     ; PRT Timer Reload Reg Chnl 0 Low
rldr0h  equ     0fh     ; PRT Timer Reload Reg Chnl 0 High
tcr     equ     10h     ; PRT Timer Control Reg
tmdr1l  equ     14h     ; PRT Timer Data Reg Chnl 1 Low
tmdr1h  equ     15h     ; PRT Timer Data Reg Chnl 1 High
rldr1l  equ     16h     ; PRT Timer Reload Reg Chnl 1 Low
rldr1h  equ     17h     ; PRT Timer Reload Reg Chnl 1 High

; DMA Controller (DMAC)
; ---------------------
sar0l   equ     20h     ; DMAC Source Address Reg Chnl 0 Low
sar0h   equ     21h     ; DMAC Source Address Reg Chnl 0 High
sar0b   equ     22h     ; DMAC Source Address Reg Chnl 0 Bank
dar0l   equ     23h     ; DMAC Dest Address Reg Chnl 0 Low
dar0h   equ     24h     ; DMAC Dest Address Reg Chnl 0 High
dar0b   equ     25h     ; DMAC Dest Address Reg Chnl 0 Bank
bcr0l   equ     26h     ; DMAC Byte Count Reg Chnl 0 Low
bcr0h   equ     27h     ; DMAC Byte Count Reg Chnl 0 High
mar1l   equ     28h     ; DMAC Memory Address Reg Chnl 1 Low
mar1h   equ     29h     ; DMAC Memory Address Reg Chnl 1 High
mar1b   equ     2ah     ; DMAC Memory Address Reg Chnl 1 Bank
iar1l   equ     2bh     ; DMAC I/O Address Reg Chnl 1 Low
iar1h   equ     2ch     ; DMAC I/O Address Reg Chnl 1 High
bcr1l   equ     2eh     ; DMAC Byte Count Reg Chnl 1 Low
bcr1h   equ     2fh     ; DMAC Byte Count Reg Chnl 1 High
dstat   equ     30h     ; DMAC Status Reg
dmode   equ     31h     ; DMAC Mode Reg
dcntl   equ     32h     ; DMAC Control Reg
; Interrupt Control Registers
; ---------------------------
il      equ     33h     ; INTERRUPT Vector Low Reg
itc     equ     34h     ; INT/TRAP Control Reg

; Refresh Control Register
; ------------------------
rcr     equ     36h     ; REFRESH Control Reg

; Memory Management Unit (MMU)
; ----------------------------
cbr     equ     38h     ; MMU Common Base Reg
bbr     equ     39h     ; MMU Bank Base Reg
cbar    equ     3ah     ; MMU Common/Bank Area Reg

; I/O Control Register
; --------------------
icr     equ     3fh     ; I/O Control Reg
```

mined by how three of the 64180's internal registers are set up.

The first register that needs to be set up is the CBAR register (Common Bank Area Register). It determines which of the three logical partitions will be used, and where they will be located within the 64k of logical address space.

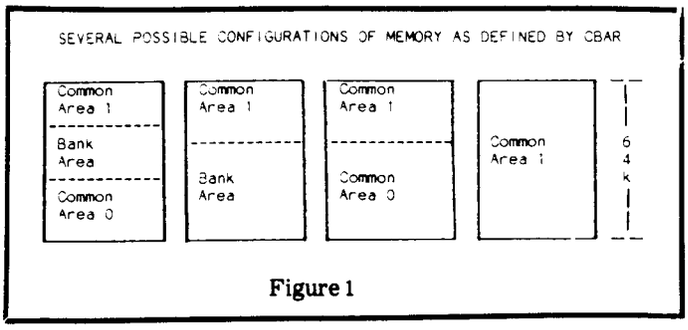


Figure 1

As you can see in Figure 1, the starting and ending addresses for the three segments WITHIN the 64k logical address space have all been set (although I didn't show the actual addresses), and whether or not a particular segment will even be used is also determined.

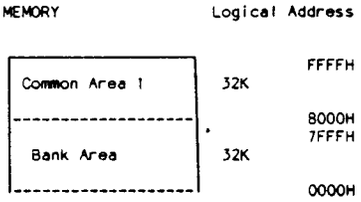
Common Area 0, if used, MUST always start at logical and physical address 0000. The Bank Area must start at either the end of Common Area 0, if it was used, or at logical address 0. Common Area 1 must start at the end of the Bank Area, if used, or the end of Common Area 0, if it was used (and the Bank Area was not used), or at address 0.

Both the Bank Area and Common Area 1 serve as areas for the actual switching of physical memory into the logical address space. All of the areas must start on a 4k physical address boundary.

Now that you have an understanding of what the 3 segments are, lets see how the CBAR register determines the actual location of the segments.

```
MMU Common/Bank Area Register (CBAR : I/O Address = 3AH)

     bit  7    6    5    4    3    2    1    0
          CA3  CA2  CA1  CA0  BA3  BA2  BA1  BA0


CA3 - CA0 (bits 7-4)
     CA  specifies the start address ( on 4K byte boundaries) for
     Common Area 1.  THIS ALSO DETERMINES THE LAST ADDRESS OF THE
     BANK AREA. After a hardware reset all of the CA bits are set
     to 1.

BA3 - BA0 (bits 3-0)
     BA specifies the start address ( on 4K byte boundaries)  for
     the Bank Area.  THIS ALSO DETERMINES THE LAST ADDRESS OF THE
     COMMON AREA 0. After a hardware reset all of the BA bits are
     set to 0.
```

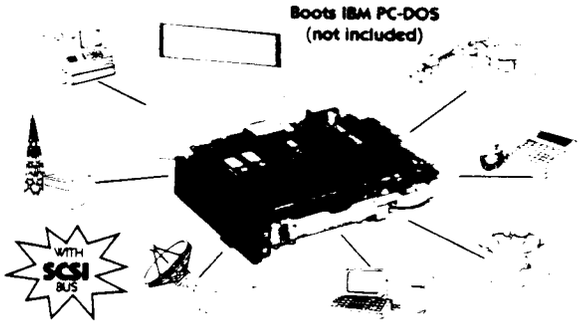
Let's actually set up the CBAR register for the following memory map, which is the same map as used in the RAM disk drive that I will show later in this article.



SCSI ENGINES

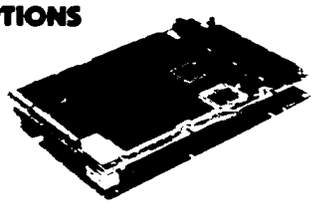
Little Board/186™ $495

High Performance, Low Cost PC-DOS Engine

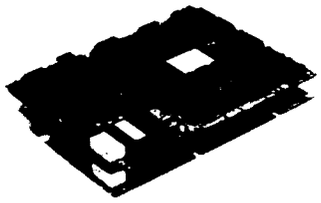


- Three times the COMPUTING POWER of a PC
- Data and File Compatible with IBM PC, runs "MS-DOS generic" programs
- 8 MHz 80186 CPU, DMA, Counter/Timers, 128/512K RAM zero wait states, 16-128K EPROM
- Mini/Micro Floppy Controller (1-4 Drives, Single/Double Density, 1-2 sided, 40/80 track)
- 2 RS232C Serial Ports (50 -38,400 baud), 1 Centronics Printer Port
- Only 5.75 x 7.75 inches, mounts directly to a 5-1/4" disk drive
- Power Requirement +5VDC at 1.25A, +12VDC at .05A; On board -12V converter
- SCSI/PLUS™ multi-master I/O expansion bus
- Software Included:
  - PC-DOS compatible ROM-BIOS boots DOS 2.x and 3.x
  - Hard Disk support

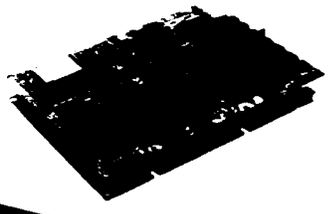
OPTIONS



PROJECT BOARD/186™ - adds 25 square inches of wire wrap prototype area with buffered and pre-decoded 80186 bus interface for Little Board/186



EXPANSION/186™ - adds five key options to Little Board/186
- 512K RAM
- 8087 co-processor
- Battery-backed Real Time Clock
- 2 RS232/422 sync, async serial ports
- I/O expansion bus



VIDEO RAM EMULATOR™ -allows use of software that writes to display controller "VIDEO RAM"

SCSI/IOP™ - permits connection of off-the-shelf STD bus industrial I/O interfaces (analog, digital, serial, display, power control, etc.)

DISTRIBUTORS

ARGENTINA: FACTORIAL, S.A., 41-0018 TLX 22408 **AUSTRALIA:** ASP MICROCOMPUTERS, (613) 500-0628, TLX 36587 **BELGIUM:** CENTRE ELECTRONIQUE LEMPEREUR, (041) 23-45-41, TLX 42621 **BRAZIL:** COMPULEADER COMPUTADORES LTDA., (41) 262-1939, TLX 416132 **CANADA:** TRI-M, (604) 438-9012 **DENMARK:** DANBIT, (03) 66 90 90, TLX 43558 **UK:** AMBAR SYSTEMS LTD., 0996 35511 TLX 837497 **FINLAND:** SYMMETRIC OY 358-0-585-322, TLX 121394 **FRANCE:** EGAL PLUS, (1) 4502-1800, TLX 680893 **ISRAEL:** ALPHA TERMINALS, LTD., (03) 49-16-95, TLX 341667 **SWEDEN:** AB AKTA, 08 54-80-80, TLX 13702 **USA:** CONTACT AMPRO COMPUTERS INC.

IBM®, IBM Corp. 80186® Intel Corp.

AMPRO COMPUTERS, INCORPORATED

67 East Evelyn Ave, • Mountain View, CA 94041 • (415) 962-0230
TELEX 4940302 • FAX (415) 962-1042

Since you want Common Area 1 to start at 8000H, you would set up the 4 most significant bits of CBAR as 1000H. To see how the value was obtained just do the following:

```
Starting address = 8000H
8000h/1000h (4k) = 8h          ;divide address by 4k
08h = 1000 bin                 ;result to binary
4 MSB's in CBAR = 1000
```

Since the Bank Area is to start at 0000h, the 4 least significant bits of CBAR will be 0's. Then just convert 10000000b to hex, and you will have a value for CBAR of 80h.

Now that the segments have been choosen for the 64k logical address space, and set up with the CBAR register, the next step is to determine what addresses in physical memory will be mapped into those segments. There are two registers used for that purpose, CBR and BBR.

```
MMU Common Base Register (CBR : I/O Address = 38H)

    bit  7    6    5    4    3    2    1    0
         -   CB6  CB5  CB4  CB3  CB2  CB1  CB0
```

CBR specifies the base address (on 4K boundaries) used to generate a 19 bit physical address for Common Area 1 accesses. After a reset all of the CBR bits are set to 0.

```
MMU Bank Base Register (BBR : I/O Address = 39H)

    bit  7    6    5    4    3    2    1    0
         -   BB6  BB5  BB4  BB3  BB2  BB1  BB0
```

BBR specifies the base address (on 4K boundaries) used to generate a 19 bit physical address for BanK Area accesses. All bits of BBR are set to 0 after a hardware reset.

As an example, suppose that you want to switch a block of memory starting at physical address 50000h into the Common Area 1 segment (at the same time switching out whatever was there). Then the CBR register would be loaded with 50h. It's that simple. You just load the register with the two most significant digits of a 5 digit hexadecimal address in physical memory.

DISK DRIVE SERVICE

5¼"..... $35
8"..... $45

SERVICE SPECIALS

Apple II Drives..... $30
Shugart SA 400/400L..... $25
Shugart SA 800/801..... $25
Shugart SA 850/851..... $35

DRIVES FOR SALE

Shugart SA 800-2 (wide frame)..... $59
Shugart SA 850 (wide frame)..... $99
MPI 52S 5¼" DS/DD full ht..... $55
Tandon 100-2 DS/DD full ht..... $70
Tandon 100-1 SS/DD full ht. (new)..... $60
Apple II Drives..... $85
Genuine "IBM" (PC) floppy contr..... $60

60 day warranty on all drives and service. Turnaround time usually 24-48 hours. Trade-in available for drives too costly to repair. Prices do not include parts or shipping. If parts are more than $20 we get permission before repairing. Units returned UPS COD unless otherwise requested. All drives for sale are reconditioned unless otherwise noted and documentation is included.

LDL ELECTRONICS

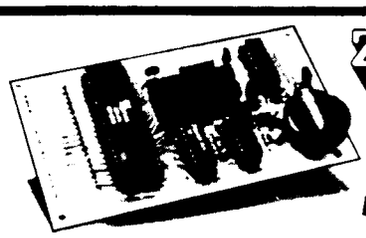
13392 158 St. N., Jupiter, FL 33478 (305) 747-7384

The BBR register is used in exactly the same manner, and since Common Area 0 always starts at physical address 0, there is no need for a register for its mapping. A few examples may help to clarify all three registers usage.

```
        CBAR = 80h      CBR = 00h       BBR = 40h

MEMORY                  Logical Address     Physical Address

                                FFFFh           0FFFFh
| Common Area 1 |       32k
|---------------|
                                8000h           08000h
                                7FFFh           47FFFh
| Bank Area     |       32k
|---------------|
                                0000h           40000h
```

Now, to switch the physical addressed memory '40000h to 47FFFh' out of the bottom 32k of logical address space, and switch the next 32k block from '48000h to 4FFFFh' into the same logical address space, just load BBR with 48h.

```
        CBAR = 80h      CBR = 00h       BBR = 48h

MEMORY                  Logical Address     Physical Address

                                FFFFh           0FFFFh
| Common Area 1 |       32k
|---------------|
                                8000h           08000h
                                7FFFh           4FFFFh
| Bank Area     |       32k
|---------------|
                                0000h           48000h
```

Ztime-I

CALENDAR/CLOCK

$69 KIT

NOW WITH FILE DATE STAMPING!

- Works with any Z-80 based computer.
- Currently being used in Ampro, Kaypro 2, 4 & 10, Morrow, Northstar, Osborne, Xerox, Zorba and many other computers.
- Piggybacks in Z80 socket.
- Uses National MM58167 clock chip, as featured in May '82 Byte.
- Battery backup keeps time with CPU power off!
- Optional software is available for file date stamping, screen time displays, etc.
- Specify computer type when ordering.
- Packages available:

Fully assembled and tested	$99.
Complete kit	$69.
Bare board and software	$29.
UPS ground shipping	$ 3.

MASTERCARD, VISA, PERSONAL CHECKS, MONEY ORDERS & C.O.D.'s ACCEPTED.

N.Y. STATE RESIDENTS ADD 8% SALES TAX

KCT

KENMORE COMPUTER TECHNOLOGIES

P.O. Box 635, Kenmore, New York 14217 (716) 877-0617

In Listing 2 you will see a portion of the BIOS that I am running on a 64180 equipped machine. It shows a RAM disk driver, and serves as a good example of some actual 64180 code that utilizes the memory past 64k. The memory on my system (a TRS-80 Model 4 with a XLR8er processor replacement board) has a 64k block of memory starting at 0000h, and then another 256k starting at 40000h.

You will notice a new instruction OUT0, and it, along with several other I/O instructions with the '0' added, are nothing more than the standard Z80 I/O instructions with the added feature of placing all 0's on the high order 8 bits of the address bus. This assures that the instruction accesses the 64180's internal ports.

You will also see the MLT instruction used. To use it just set up a register pair (BC,DE,HL, or SP) with the values to be multiplied (an 8 bit value in each 8 bit register). The result will be stored as a 16 bit value in the same register pair.

I did not include the Disk Parameter Header nor the Disk Parameter Block for the RAM disk, as they are nothing more than the standard CP/M structures. The memory maps used as examples in the prior text are the same ones that this RAM disk driver uses. The routines are strictly the low level drivers, and the high level disk routines are not shown.

There are already several packages that are coded to allow the programmer to access the 64180's extra memory from within a high level language, without having to worry about the complexities of addressing it at the chip level. They are Borland's Modula-2 from Micro-Mint, and 64180 Basic from Softaid.

64180 Basic is currently available, and Modula-2 will soon be. There has also been talk of several new CP/M packages being written specifically for the 64180, one of them being a Lotus-like spreadsheet. Echelon is currently working on a new multi-tasking DOS and banked ZCPR3 that will both utilize the 64180's memory addressing features.

Listing 2

```
;  Memory drive read routine
;     Input:  Select parameters in Select Control Block
;     Output: Record moved to (DSBDMA)

mdread: cp      02h             ; Is it a write
        jr      z,mdwrit        ; Jump if so
        call    mdaddr          ; Else set up addresses
        jr      mdmove          ; And read the record

;  Memory drive write routine
;     Output: Record moved from (DSBDMA)

mdwrit: call    mdaddr          ; Set up addresses
        ex      de,hl           ; Switch for write

;  Memory drive data move routine
;     Input:  A=Address select bits for move
;             HL=Source address for move
;             DE=Destination address for move
;     Output: 128 bytes moved as requested

mdmove: di                      ; No interrupts
        out0    (bbr),a         ; Switch in bank
        call    movrec          ; Move the record
        xor     a               ; 0=bank 0
        out0    (bbr),a         ; Switch in normal map
        ei
        ret

;  Memory drive address setup routine
;     Input:  Information in Select Control Block
;     Output: A=Map address select bits
;             DE=Internal record buffer address
;             HL=Record address in alternate Memory Map

mdaddr: push    hl              ; Save buffer address
        ld      l,e             ; Sector # to L
        ld      h,128           ; Multiply by 128
        mlt     hl              ; HL now contains address in bank
        ld      a,c             ; Track # to A
        rlc     a               ; Multiply by 8
        rlc     a
        rlc     a
mdadr1: add     a,bank1         ; Add BBR offset
        pop     de              ; DMA addr to DE
        ret
```

The Hitachi 64180 holds great promise for the 8-bit world, and for CP/M in particular. If half of the products promised for the chip are delivered, then you you will find 64180 based systems will be around for a long time.

If you are interested in more information on the XLR8er, a 64180 based enhancement for the TRS-80 Model 4, contact H.I. Tech Inc., P.O. Box 25404, Houston, TX 77265, 1-800-835-2246 ext 202, or contact my BBS system, Rio Grande Z-Node #39, 915-592-4976. ■

Z sets you free!

WHO WE ARE
Echelon is a unique company, oriented exclusively toward your CP/M-compatible computer. Echelon offers top quality software at extremely low prices; our customers are overwhelmed at the amount of software they receive when buying our products. For example, the Z-Com product comes with approximately 80 utility programs; and our TERM III communications package runs to a full megabyte of files. This is real value for your software dollar.

ZCPR3
Echelon is famous for our operating systems products. ZCPR3, our CP/M enhancement, was written by a software professional who wanted to add features normally found in minicomputer and mainframe operating systems to his home computer. He succeeded wonderfully, and ZCPR3 has become the environment of choice for "power" CP/M users.

Multiple Commands per Line
You can easily use multiple commands per line under ZCPR3. Simply separate the individual commands with semicolons. For example, "PIP B:=A:*.TXT;STAT B:*.*" will copy files and then show you the STAT results.

User-Programmed menu systems
ZCPR3 comes with three different menu systems that you can use to create custom menu-driven "front ends" for your computer. This is especially useful for setting up menus for your spouse or co-workers to use the computer, as they never have to see the A> prompt. All they have to do is press a single key to run any single or multiple CP/M programs, and when the task is done, control is automatically returned to the menu (ordinary CP/M menu programs cannot do this).

Extended Command Processing
When you type a command under CP/M, it will only look for the program in the current drive and user area. ZCPR3 gives you more flexibility by additionally searching other disks and user areas when resolving commands. You have full control of this function, called the PATH. This is probably the one element of ZCPR3 that is missed most if you return to "ordinary" CP/M.

Also, ZCPR3 supports the capability of grouping all your commonly used utility programs into a library file (*.LBR). This is great for systems with a small number of directory entries per disk, as the library file only uses one entry. It also has the advantage of reducing disk space requirements for a given set of programs, allowing you to put more programs on a disk. And the programs in the library file are invokable from the command line just like any other program not in the library.

Other Features
There's much more to ZCPR3, like named directories, online help system, etc., but it can't be described on one page. If you would like more information, consider the books shown below.

Z-SYSTEM
Perhaps the only shortcoming of ZCPR3 is that it is not a complete replacement for CP/M. This is what the Z-System does. The Z-System contains ZCPR3 and an additional module, ZRDOS, and is a complete replacement for CP/M. ZRDOS adds even more utility programs, and has the nice feature of no need to warm boot (^C) after changing a disk. Hard disk users can take advantage of ZRDOS "archive" status file handling to make incremental backup fast and easy. Because ZRDOS is written to take full advantage of the Z80, it executes faster than ordinary CP/M and can improve your system's performance by up to 10%.

INSTALLING ZCPR3/Z-SYSTEM
Echelon offers ZCPR3/Z-System in many different forms. For $44 you get the complete source code to ZCPR3 and the installation files. However, this takes some experience with assembly language programming to get running, as you must perform the installation yourself.

For users who are not qualified in assembly language programming, Echelon offers our "auto-install" products. Z-Com is our 100% complete Z-System which even a monkey can install, because it installs itself. Z-Com includes many interesting utility programs, like UNERASE, MENU, VFILER, and much more.

Echelon also offers "bootable" disks for some CP/M computers, which require absolutely no installation, and are capable of reconfiguration to change ZCPR3's memory requirements. At present, only Kaypro computers have this option available.

BOOKS
We sometimes joke around the office that we are really in the business of publishing, not selling software. We have books. Lots of books. We have to have lots of books, considering how powerful our software is and the large quantity of different packages we offer. Here are our best sellers:

ZCPR3: The Manual
This is the "bible" for the ZCPR3 user. An exhaustive technical reference, bound softcover, 350 pages. Contains descriptions of each ZCPR3 utility program, a detailed discussion about the innards of ZCPR3, and a full installation manual for those doing their own installation. You could order it from B. Dalton, but why? Get it from us.

The Z-System User's Guide
For those who are not technically inclined. This is an excellent tutorial-style manual filled with examples of how to use the power of ZCPR3/Z-System most effectively, written by two highly experienced Z users. (One user is a lawyer, the other a writer; this proves that anyone can use Z and benefit from it.)

ZCPR3: The Libraries
The extensive documentation for the libraries of ZCPR3, SYSLIB, Z3LIB, and VLIB. A must for any serious user of these programming tools. Loose-leaf notebook style; easy to work with as it will lay flat on your desk.

THERE'S MORE
We couldn't fit all Echelon has to offer on a single page (you see how small this type is). We haven't begun to talk about the many additional software packages and publications we offer. Send in the order form below and just check the "Requesting Literature" box for more information.

Item	Name	Price
1	ZCPR3 Core Installation Package	$44.00 (3 disks)*
2	ZCPR3 Utilities Package	$89.00 (9 disks)
3	Z3-Dot-Com (Auto-Install ZCPR3)	$99.00 (6 disks)*
4	Z3-Dot-Com "Bare Minimum"	$49.95 (1 disk)
5	Z-Com (Auto-Install Z-System)	$119.00 (7 disks)*
6	Z-Com "Bare Minimum"	$69.95 (2 disks)
12	PUBLIC ZRDOS Plus (by itself)	$59.50 (1 disk)
13	Kaypro Z-System Bootable Disk	$69.95 (3 disks)
20	ZAS/ZLINK Macro Assembler and Linker	$69.00 (1 disk)
21	ZDM Debugger for 8080/Z80/HD64180 CPU's	$50.00 (1 disk)
22	Translators for Assembler Source Code	$51.00 (1 disk)
23	REVAS3/4 Disassembler	$90.00 (1 disk)
24	**Special** – Items 20 through 23	$150.00 (4 disks)
25	DSD-80 Full Screen Debugger	$129.95 (1 disk)
27	The Libraries: SYSLIB, Z3LIB and VLIB	$69.00 (8 disks)
28	Graphics and Windows Libraries	$49.00 (1 disk)
29	**Special** – Items 27, 28, and 82	$129.00 (9 disks)
40	Input/Output Recorder IOP (I/OR)	$39.95 (1 disk)
41	Background Printer IOP (BPrinter)	$39.95 (1 disk)
42	Programmable Key IOP (PKey)	$39.95 (1 disk)
43	**Special** – Items 40 through 42	$89.95 (3 disks)
60	DISCAT Disk cataloging system	$39.99 (1 disk)
61	TERM3 Communications System	$99.00 (6 disks)
64	Z-Msg Message Handling System	$99.00 (1 disk)
81	**ZCPR3: The Manual** bound, 350 pages	$19.95
82	**ZCPR3: The Libraries** 310 pages	$29.95
83	Z-NEWS Newsletter, 1 yr subscription	$24.00
84	**ZCPR3 and IOPs** 50 pages	$9.95
85	**ZRDOS Programmers's Manual** 35 pages	$8.95
88	**Z-System User's Guide** 80 page tutorial	$14.95

*Includes **ZCPR3: The Manual**

Echelon, Inc.
885 N. San Antonio Road, Los Altos, CA 94022 USA
415/948-3820 (order line and tech support)

NAME _____
ADDRESS _____
_____
TELEPHONE _____ DISK FORMAT _____
☐ REQUESTING LITERATURE

ORDER FORM

Payment to be made by:
☐ Cash
☐ Check
☐ Money Order
☐ UPS COD
☐ Mastercard/Visa:
_____
Exp. Date _____

California residents add 7% sales tax.
Add $4.00 shipping/handling.

ITEM	PRICE
_____	_____
_____	_____
_____	_____
Subtotal	_____
Sales Tax	_____
Shipping/Handling	_____
Total	_____

ZSIG Corner

Command Line Generators and Aliases

by Jay Sage, ZSIG Software Librarian

When I first offered to write a regular column for The Computer Journal, I wondered if I would have enough material for a column every two months. Instead, the problem has become one of finding the time to set down all the thoughts I have. For this issue I had planned to cover four topics, there is only room for three of them. They are: 1) corrections of some errors in the last column (and an excuse for more discussion of flow control in Z System); 2) a brief rundown on the files in the first officially released ZSIG diskette; and 3) a discussion of command-line-building programs in ZCPR3, including aliases and shells.

Corrections

There were two errors that I noticed in the last column. One was a minor one that was Art Carlson's fault; the other was more serious and was my doing. First for the easy one. Art was kind enough to add information at the end of the column on how to contact me, but the Z-Node he listed there was not mine. It was the Lillipute Z-Node in Chicago, which is the official ZSIG remote access system (its phone number is 312-649-1730). Messages left for me there will get to me, but I will get them sooner if they are left for me on my own node in Boston at 617-965-7259. I can also be reached on my voice phone at 617-965-3552 (please don't mix the two numbers up, especially if you are calling in the middle of the night!). Finding me at home is not always easy, and your chances will be best if you call between 10 p.m. and just after 11 p.m. (Boston time, of course). If you would rather write, the address is Jay Sage, 1435 Centre Street, Newton Centre, MA 02159.

Now for the error I made. In the discussion of the flow control package (FCP) and the transient IF.COM, I said that one can force use of the transient program by including a DU: or DIR: prefix or even just a colon in front of the 'IF". This is not true. I thought I had verified the statement experimentally, but my experiment was flawed, and the conclusion incorrect. A colon does force the command processor to skip commands in the resident command package (RCP) and in the CPR itself and to proceed with a search for a COM file, but all the FCP resident commands are intercepted no matter whether there is a colon or not. There is a very good reason for this. The FCP commands must be executed even when the current IF state is false. This is especially clear for commands like ELSE, which reverses the current IF state, and FI, which terminates the current IF state. Transients ELSE.COM and FI.COM could not do this. After a bit of thought you can probably see that this is true for all the flow control commands.

I have long been searching for ways to give the user control over whether IF processing is performed by resident or transient code. One solution that I introduced some time ago was adding alternative names for condition options in IF.COM so that one could force the more powerful transient processing to be performed even when the function was supported by the resident FCP. The trick was to use condition names in IF.COM that were different from the names of the same functions in the FCP module. Specifically, I changed the first letters to 'X' (for example XXIST for EXIST and XNPUT for INPUT). Frank Gaude' in the Echelon Z-News has reported examples in which the transient program is given a name other than IF.COM (such as IF13.COM). Then the command "IF INPUT" gives resident processing and "IF13 INPUT" transient processing. This will work correctly for first level IF processing but will not work in general.

Neither of these solutions was really satisfactory. A proper solution, I felt, would operate both correctly and automatically. I have now written a new FCP package (FCP10) that is presently under test by ZSIG program committee members and should be ready for release with the next ZSIG diskette. It handles both resident and transient code. It can examine the command line to see if there was a colon before the IF. In that case, the FCP ignores its internal condition options and invokes the transient IF.COM immediately. If no colon was present, the FCP first looks to see if the condition option is included in the resident code. If not, it automatically invokes the transient IF processor. The user need not be concerned with which options are resident in the FCP. The script "IF NULL $1" in an alias will take advantage of fast resident processing if the NULL option is supported in the resident code or will automatically invoke transient processing if not. When one wants to be sure to get transient processing, one simply uses the colon as in ":IF EXIST FILE1,FILE2".

To go along with FCP10 there is a new transient I processor, COMIF10. It supports dozens of additional conditic tests. One is AMBIG, which tests a file specification for ar biguity (question marks or asterisks). Register (numeric) ar string (alphabetic) comparisons are now extended to the fu range of tests: equal, not equal, greater, less, greater than (equal, and less than or equal. I plan to add file attribute testin (SYS, DIR, RO, RW, ARC, WP) and testing for the presence of Plu*Perfect Software's exciting Backgrounder task-swapping program.

The new FCP/COMIF combination also adds an important new twist in transient processing. They have the option of loading the transient code not at 100H, as has been done until now, but high in memory, where it will not overwrite a user program that is loaded at 100H. In this way the GO command can be used to rerun the last user program after flow control processing no matter whether resident or transient flow processing was used. Thus the user need not concern himself with how the flow processing was performed.

While I was at it, I also added two new commands to the FCP module. One, IFQ (if query), is designed to help users — advanced as well as novice — learn how flow control works (or,

perhaps, is not working as one intends). It displays on the screen the complete flow state of the system — the true/false status of all IF levels. The second is ZIF (zero ifs). It is like XIF (end all ifs) except for one thing. XIF clears all IF states only if the current IF state is true; if the current IF state is false, XIF is flushed (ignored) just like any other command. ZIF, on the other hand, clears all IF states no matter what. I'm sure that everyone who has experimented with flow control has at one time or another gotten himself so messed up that nothing seemed to work. The only way out, short of rebooting, was to type a string of FIs until things started working again. ZIF is a quick way to reinitialize the flow control system (and it takes very little code in the FCP).

ZSIG Diskette #1

Now that I have atoned, I hope, for my sin in the last issue, I will turn to the first new subject, the inaugural ZSIG diskette. Let me remind you that this diskette and a number of others put out by NAOG/ZSIG (NAOG = North American One-Eighty Group, the group formed to support the SB180 computer and other computers using the Hitachi HD64180 microprocessor) can be ordered from NAOG/ZSIG (P.O. Box 2781, Warminster, PA 18974). I encourage all of you to join ZSIG ($15). You'll get a nice newletter with Z System tips and details on all the NAOG/ZSIG diskettes.

Here is a listing of the files in ZSIG diskette #1.

Z-RIP.LBR VERROR17.LBR VCED18.LBR

ZCRCK.LBR ZFINDU.LBR ZLDIR.LBR
ZTXTTOWS.LBR ZWC.LBR

PPIP14.LBR UF.LBR

LDSK20.LBR W20.LBR

The first three programs are by Paul Pomerleau. Paul is a speed freak and, like many of us, found the process of installing Z programs with Z3INS (the standard installation utility) tedious and slow. So Paul wrote Z-RIP, given that name because it rips through an entire disk of files at incredible speed, automatically identifying the ZCPR programs and installing them. The new autoinstall versions of ZCPR3 may make Z-RIP (not to mention Z3INS) obsolete, but it is great for those running standard ZCPR. VERROR is Paul's video error handler. It provides a screen display of the entire command line in which an error was detected and allows the user to edit it freely, moving about using WordStar-like commands. VCED is Paul's Video Command EDitor, a video history shell. With VCED running as a shell, the user always has full command-line editing. In addition, past commands can be recalled, searched for, edited, and run. As if that were not enough, it doubles as a video error handler as well! Paul astutely noted the functional similarity between correcting old commands with errors and entering new commands — so he combined the two functions in a single program.

The next group of five programs comprises Z versions of common CP/M utilities. Most of these were created by the prolific program fixer and NAOG/ZSIG chief, Bruce Morgen. The main feature that makes these programs ZCPR3-compatible is their ability to accept named directory (DIR:) references as well as drive/user (DU:) references. For programmers and aspiring programmers reading this, you should know that the code to do this in ZCPR3 is actually much simpler than the CP/M code needed just to recognize the DU: form. This is because the ZCPR3 command processor already does all the work for the first two arguments on the command line (including translating named directory references into drive/user values). Unlike CP/M, ZCPR3 saves not only the drive but also the user number in the default file control blocks at 5CH and 6CH. A ZCPR3 program need only fetch the values from the appropriate locations. The hardest part of making these ZCPR3 versions of the CP/M programs was stripping out the complex and lengthy parsers required to accept DU: syntax in CP/M. (So much for the myth of ZCPR3 complexity! Programming in ZCPR3 is often, as in this example, simpler than programming in standard CP/M.)

The third group of programs includes two more ZCPR3 versions of CP/M programs. They are listed separately only because they do not have names with Zs in front! Here is a quick listing of the functions of all seven of these converted programs:

ZCRCK — computes cyclic redundancy check codes for files using both common CRC polynomials

ZFINDU — searches for text strings in files, including files that are squeezed

ZLDIR — displays a directory of the files in a library

ZTXTTOWS — converts standard text files to WordStar files

ZWC — counts the number of words in a text file

PPIP14 — copies files (as does PIP) but with nicer interface and fast — I renamed it to COPY and use it all the time

UF — Steven Greenberg's ultrafast file unsqueezer

The last two programs are original creations for the Z System. LDSK, by Wilson Bent with modifications by Earl Boone, solves a longstanding problem that owners of floppy-disk-based computers had with named directories. With hard disks, there is an unchanging association between directory names and drive/user values, but with floppies the association changes every time the diskette is changed. Wilson devised this nifty scheme for automatically loading the named directory register (NDR) with the names associated with user areas on a floppy diskette. To give a user area a name, one simply puts a (usually zero-length) file in that user area with a name of the form "-NAME". When LDSK is run (specifying the drive to be loaded), it scans the disk for files of this type, strips the leading hyphen, and creates an entry in the NDR associating the name with that user number on the drive. As I wrote in the last column, I still have a lot of floppy-only systems, and I love LDSK.

Haven't you at times wished that you could take some program that only works on a single file and magically make it work with an ambiguous file reference. Well, Steve Cohen did, so out of his programmer's hat he pulled the wildcard shell 'W' to do it. It just shows again that the only real limitation with the Z System is one's imagination! Here are some examples of how 'W' can be used. Bob Freed wrote a quick little program called PCPCK that checks a file for proper transmission over Telenet's PC-Pursuit packet network (certain character sequences cause problems). The trouble is, PCPCK only works on a single file, and it is no fun to run it manually on every file one is about to send somewhere. But along comes 'W' and all I have to do is enter "W PCPCK *.*" and away we go. Or suppose you are just lazy and hate typing exact names of files. Just put a 'W' in front of the command and enter a wildcard file name that specifies the file you want. That's all there is to it. I have 'W' implemen-

ted in an alias on my Z-Node system so that users can type a file without having to enter the exact name. If a user can't remember (or doesn't really care) whether the file is AUTOINST.FIX or AUTOINST.FQZ or AUTOINST.FZX, all he has to enter is "TYPE AUTO*.*" and the file (whatever it is called) will appear on the screen.

Command Line Generators

Many people call me about problems they are having getting an alias or VFILER script to work correctly. Often the problem turns out to be a misunderstanding of what command line generators are really doing. I will try to shed a little more light on that subject here.

First a little philosophy. There are many features in the Z System about which one might well at first just shrug one's shoulders and say, "So what!" The flow control system discussed earlier is one such feature, and multiple commands on a line might be another. After all, how many of us actually think far enough ahead to enter more than one command at a time anyway? Well, the answer lies in the interplay of all the features in Z System and in the ways they allow things to be accomplished automatically.

Aliases

The multiple command capability of Z System, for example, is important not so much because it allows the user to enter a whole sequence of commands manually but rather because it allows other programs to do so automatically. The simple, standalone 'alias' created with the original ALIAS.COM or one of the more sophisticated alias programs like TALIAS, BALIAS, or VALIAS is a good example. Let's see how such an alias might be used. Suppose we are working on a new program with a source file called MYPROG.Z80. Our standard sequence of operations is to edit the source with a command like "EDIT MYPROG.Z80" and then to assemble it with a command like "ASM MYPROG.AAZ" and then to load it with a command like "MLOAD MYPROG1". We can speed things up and reduce the amount of typing (and the number of typos!) by creating an alias which we might give the name DO.COM. We would create it, with VALIAS for example, with the following script (command line form):

```
EDIT MYPROG.Z80;ASM MYPROG.AAZ;MLOAD MYPROG
```

Now when we want to start a new cycle, we just enter the easily spelled command "DO". The rest is automatic.

But how does this alias actually work? When you enter the command "DO", the operating system loads DO.COM into memory and starts running it. DO contains within its file the script line put there by VALIAS.COM (for example) when the alias was created. DO.COM has code to determine where the Z System multiple command line is located in memory (this information comes from what is called the environment descriptor, whose address is installed in a standard location near the beginning of all true Z System programs). Next DO.COM takes its command script, appends any other commands in the multiple command line that come after the "DO" command, and then writes the result back to the command line buffer. When it then returns to Z System, the ZCPR3 command processor, as usual, looks at the command line buffer to see if there are more jobs listed there for it to do. Since DO.COM has filled the command line buffer with the script, ZCPR3 responds just as if we had typed the long command line script instead of the simple "DO".

SAGE MICROSYSTEMS EAST

Selling & Supporting
The Best in 8-Bit Software

- **Plu*Perfect Systems**
  - Backgrounder II: switch between two or three running tasks under CP/M ($75)
  - DateStamper: stamp your CP/M files with creation, modification, and access times ($49)
- **Echelon (Z-System Software)**
  - ZCOM: automatically installing full Z-System ($70 basic package, or $119 with all utilities on disk)
  - ZRDOS: enhanced disk operating system, automatic disk logging and backup ($59.50)
  - DSD: the incredible Dynamic Screen Debugger lets you really see programs run ($130)
- **SLR Systems (The Ultimate Assembly Language Tools)**
  - Assemblers: Z80ASM (Z80), SLR180 (HD64180), SLRMAC (8080), and SLR085 (8085)
  - Linker: SLRNK
  - Memory-based versions ($50)
  - Virtual memory versions ($195)
- **NightOwl (Advanced Telecommunications)**
  - MEX-Plus: automated modem operation ($60)
  - Terminal Emulators: VT100, TVI925, DG100 ($30)

Same-day shipping of most products with modem download and support available. Shipping and handling $4 per order. Specify format. Check, VISA, or MasterCard.

Sage Microsystems East
1435 Centre St., Newton, MA 02159
Voice: 617-965-3552 (9:00 a.m. - 11:15 p.m.)
Modem: 617-965-7259 (24 hr., 300/1200/2400 bps, password = DDT, on PC-Pursuit)

Now let's see how flow control can be used with alias scripts. One problem with the command sequence in our example arises when the assembler reports an error. In that case there is no sense going through the MLOAD operation. Assemblers like ZAS from Echelon and Z80ASM from SLR Systems set a flag in the Z System to show whether or not they encountered any fatal errors during the assembly, and the flow control command "IF ERROR" can test the state of that flag. We can improve ou script as follows:

```
EDIT MYPROG.Z80;ZAS MYPROG;IF ~ ERROR;MLOAD MYPROG;FI
```

In this script the MLOAD command will only be executed if th program error flag has not been set by ZAS (the tilde '~' has th meaning 'not'). Typing all those flow control commands manually would be more trouble than entering single commands at a time, but with an alias we are still typing only two letters: "DO".

So far so good. But what happens when we want to start work on another program, say NEWPROG? Do we have to create a new alias, such as DONEW? The answer is that the alias program actually does much more than just copy a command script as is into the multiple command line buffer. It is capable of making parameter expansions, the simpler examples of which are like the parameter expansions that occur with the CP/M SUBMIT program. We can store the alias script as:

```
EDIT $1.Z80;ZAS $1;IF ~ERROR;MLOAD $1;FI
```

The '$1' is a symbol representing the first token after the command on the command line that invoked the alias program. Thus when we enter the command "DO MYPROG" we get the first script we discussed, but when we enter "DO NEWPROG" we get a command line for working on NEWPROG instead. A single alias thus becomes very flexible. There are quite a number of parameter forms that can be processed by aliases, and I refer you to Rick Conn's "ZCPR3, The Manual" and various HELP files for more detailed information.

Now let's try something a little trickier. Sometimes we have already edited a file and just want to assemble and load it (if there is no error in assembling, of course). So we create an alias called AL (for assemble/link)

```
ZAS $1;IF ~ ERROR;MLOAD $1;FI
```

[I am using ZAS in these examples rather than the SLR Z80ASM, which I prefer, because the SLR assemblers can produce a COM file directly in one pass and do not need MLOAD or the flow control error checking. Thus they do not serve the purposes of my example here.] Now what do you think will happen if we define our DO alias as follows:

```
EDIT $1.Z80;AL $1
```

Do you think that will work? One alias inside another? Well, it will indeed! Aliases can be nested. How deeply? Without any limit! Before we explain why this is, let's look at an even more fascinating example. When I sit down to work on a program, I typically go through one edit/assemble cycle after another (just don't seem to be able to get it right the first time). So I make my DO alias have the following script:

```
EDIT $1.Z80;AL $1;DO $1
```

This alias actually invokes itself!! When one cycle is finished, it just goes back for more. Now let's look at what goes on in the system after we enter the command "DO MYPROG". The DO alias expands its script and writes the following command line into the multiple command line buffer:

```
EDIT MYPROG.Z80;AL MYPROG;DO MYPROG.
```

After the editing is finished, AL runs, expands its script, and fills the command line buffer with the following command line:

```
ZAS MYPROG;IF ~ ERROR;MLOAD MYPROG;FI;DO MYPROG
```

Note that the alias always appends to its own script any other commands in the command line after itself (hence the DO MYPROG on the end). Now ZAS runs, and, depending on whether there were errors or not, MLOAD may run. Finally ZCPR3 gets to the DO command, and we are right back where we started. The whole process is repeated (and repeated again). In fact, the only trouble with this alias is that there is no way out! You can't stop!

Well, we all hope we will get the program right eventually, so we really would like to be able to get out of the alias. Flow control can help us again. Consider the script

```
EDIT $1.Z80;AL $1;ECHO EDIT AGAIN?;IF INPUT;DO $1;FI
```

Now, before reinvoking DO, the alias asks us if we want to edit the file again. If we give a negative answer (anything other than carriage return, space bar, 'Y' for yes, or 'T' for true), the loop is broken. If we answer affirmatively with a quick tap of the return key, we start again. Very quick and easy.

There is one subtle problem, however. If you go through the exercise of expanding the alias scripts, you will see that with each cycle an extra 'FI' builds up at the end of the command line. Even more careful analysis will show that with each cycle we go one IF level deeper as well. One of two problems will eventually spoil our plan. Either the command line will get so long that it won't fit in the command line buffer, or we will run out of IF levels (eight is the maximum). What can we do about these problems?

The FCP has the XIF command precisely for this reason. If we put an XIF command at the beginning of the script, we will reset the IF system to level 0 every time we reenter the loop. Then the limit will be overflow of the command line. When this happened to me, I invented a special type of alias — the recursive alias — and incorporated it into my VALIAS program (as far as I know only VALIAS and ARUNZ support this alias type). It works the same as a regular alias except for one thing — it does not append to the script expansion any commands that were pending in the command line buffer; it just throws them away. Thus in the above example all the FIs would be discarded when DO was invoked again, and the pileup would be avoided. When an alias is created with VALIAS, one can select either a normal alias or a recursive alias. But note that no other command can ever follow a recursive alias on a multiple command line. Recursive aliases should be used only in special cases like the one described here.

Shells

Aliases are not the only command line generators. Most shells also generate command lines for the user. In some cases (VCED, described above, and MENU) this is their main purpose; in other cases it is secondary (VFILER). Before we examine the way they generate command lines, let's look at the way shells operate in the Z System.

The essential purpose of shells is to create just the kind of recursive command situation we were just developing with our alias example. But they achieve that function in a very different way. A shell has a kind of schizophrenic personality as a result of being invoked in two significantly different circumstances. One circumstance is when it is invoked directly or indirectly (e.g., from an alias) as the result of a user command. In this case, the shell has one basic purpose -- to perpetuate its own existence as a command. It does this by entering its name as a command into a special buffer (area in memory) in the Z System called the shell stack. Having done that, it can then return control to the operating system. (The smarter shells generally do something a little more sophisticated at this point, but the principle is correct as I have described it.)

Now we come to the unique role of shells in the Z System. The CP/M command processor gets commands from only two possible sources: 1) from a submit file, if one exists, or 2) from the user. The Z System gets commands from at least four sources and in the following order of priority (ignoring the tricky role of ZEX): 1) from the multiple command line buffer; 2) from a submit file; 3) from the shell stack; and 4) if all else fails, from the user. Observe that so long as the shell stack has a command in it, the command processor will never turn to the

user for input! That is why one can regard the shell as taking over the command processor function. While the shell is running, it becomes the source of commands for the system.

How does the shell do this? By expressing its second and more dramatic personality. Another special buffer in the Z System, the message buffer, contains a flag byte that is set by the ZCPR3 command processor to indicate whether a program has been invoked as a user command or as a shell (or as an error handler). We have already discussed the simple goal of the shell in the former case. In the latter case the shell actually carries out its real function in life. Let's consider the MENU shell as an example.

When the MENU.COM is loaded as a shell, it displays a screen of command choices to the user. Each single-character choice is associated with a command line script, much like the alias script. When the user strikes a key, MENU looks up the script associated with that character, expands the script (substituting parameters), and puts the resulting command into the multiple command line buffer. It then returns control to the ZCPR3 command processor. ZCPR3 executes the commands in the command buffer one after another until they have all been performed. Then, when the command buffer is empty again, ZCPR3 looks in the shell stack, finds the MENU command there, and runs MENU again. This process continues until a special user key is entered (control-c in the case of MENU) that signals the shell that it should remove itself from the shell stack. Then things return to the state they were in before the shell was invoked initially by the user. Shells, by the way, can be nested (the usual shell stack is four entries deep), so when one shell removes itself from the shell stack, control may still not return to the user. Another shell, whose role was superceded by the most recent shell, may now come back into control.

With the MENU.COM the displayed menu of choices and the scripts associated with the choices are both included in a text file that is read in by the program. This makes it very easy for the user to create and modify the display and the scripts. Considering again our program development example, we might create a menu screen with the following display:

```
S. Select name of program
E. Edit program source code
A. Assemble program to HEX file
L. Load program to COM file
R. Run program
F. Full cycle (edit, assemble, load)
```

These choices might be associated with the following command scripts:

```
S setfile 1 "Enter name (only) of program to work on: "
E edit $n1.Z80
A zas $n1
L mload $n1
R $n1
F edit $n1.z80;zas $n1;if ~er;mload $n1;fi
```

There are two interesting new parameters illustrated in these scripts. One is the $N1 parameter. As part of the Z System environment buffer, four system file names are defined. MENU can read these four file names and put into scripts the complete filename ($Fn), the name only ($Nn), or the type only ($Tn), where 'n' is 1, 2, 3, or 4. The 'S' selection sets the first system file name using the program SETFILE, and the others then use it.

The 'S' selection illustrates the other new script parameter — prompted user input. When the script for choice 'S' is being expanded, the text between quotes will be displayed as a prompt to the user, and one line of user input will be substituted into the command line in place of the prompt. It is with prompted input that many users get confused and make mistakes. Suppose you want to be clever and helpful by displaying a directory of existing programs to jog the user's memory before asking for his choice. You might think of using the script

```
S dir *.z80;setfile 1 "Enter program name: "
```

This will not have the effect intended! One must not forget that the user is prompted for input by the shell **at the time the script is expanded**, not at the time when the command line is executed. In this example the user will be prompted for his choice before the directory is displayed. Thus, the directory display is useless.

To achieve the result intended above you must combine scripts. You can create an ARUNZ alias called SETNAME with the following script (ARUNZ supports prompted input):

```
SETNAME setfile 1 "Enter name of file: "
```

The MENU script would then be:

```
S dir *.z80;arunz setname
```

When the MENU script is expanded, the command becomes "DIR *.Z80;ARUNZ SETNAME", and this command is then run. It is not until ARUNZ SETNAME is executed that ARUNZ prompts the user for the name of the file. At this point the directory of files with type Z80 has already been displayed on the screen.

There is obviously much more that could be said about the command line generators in ZCPR3. The discussion here has been only an overview, but I hope that it will inspire you to take a fresh look at and to experiment with aliases and shells of all kinds: the standalone aliases generated by ALIAS, VALIAS, TALIAS, or BALIAS; the text-file-based alias generator ARUNZ with its ALIAS.CMD file; the menu- or macro-type shells MENU, VMENU, FMANAGER, VFILER, and ZFILER; and the command-line history shells HSH and VCED.

Plans for Next Time

As I said at the beginning of the article, I had planned t cover, along with all the subjects above, techniques fo customizing the Z-COM self-installing version of Z System. Bu there just isn't the time or space. So I will have to leave that fo the next issue. Let me just say one thing here. If you do no already have Z System running on your computer and have held back on buying Z-COM from Echelon because you thought it would not offer you the flexibility of a custom installation, hold off no longer. Buy Z-COM! Start the exhilarating process of discovering Z System now. By the time my discussion of Z-COM hacking is complete, you will know how to get just as much flexibility with Z-COM as with a manually installed version. It is much more fun to start with something that is working and to improve it than it is to spend many frustrating weeks trying to get an initial manual version working.

I want to close with my usual invitation and encouragement — please write and call with your comments and suggestions of all kinds. ■

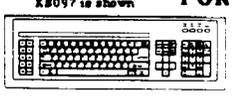
$45 MORROW CP/M ENGINE
- Over 15,000 of these single board computers installed.
- Free copy of the "MORROW OWNERS REVIEW". The national magazine devoted just to the MORROW computer. while supplies last
- We have over 2000 in stock
- 4 mhz Z80A CPU, 64K RAM 16/32K ROM
- 2 RS232 serial ports (300-19.2k baud with db25 connectors installed)
- Centronics printer port cable extra $8
- Power requirements- +12vdc -12vdc +5vdc
- Floppy disk controller up to 4 drives. SSDD standard. Rom for DSDD 48tpi or 96tpi $12 extra including assembled bios
- CP/M bios, Wordstar, and Bazic included
- Schematic, bios.asm, maint manual and users guide included
- Copy program to read/write non morrow formats such as Kaypro, Osborne, Xerox, & etc.
- Optional rs232 terminal pcb, TTL monitor pcb, and display tube $59 (with purchase of cpu, a 30% savings from seperate price)

Call fo a copy of 15 day trial agreement. Tax & frieght extra. Send check or add 1.90 for COD. Price may change. Store price may differ. While supplies last. No POs, terms, or credit cards

Silicon Valley Surplus
415-261-4506
4401 OAKPORT OAKLAND CA, 94601
OPEN 10am-6pm
CLOSED SUN & MON

SPECIAL PURCHASE

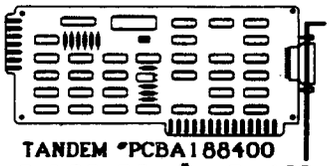
Multitech INDUSTRIAL CORP KB097 is shown
KEYBOARDS FOR IBM® A.T.



Both work on the PC or the AT. If you don't like the feel return it for a refund 15 day trial period see below

84 KEYS * KB084 ... $39.00
97 KEYS * KB097 ... $49.00

FLOPPY CONTROLLER WITH LPT1 PRINTER PORT FOR IBM® OR CLONES



TANDEM *PCBA188400

ONLY $ 24.99

15 DAY TRIAL PERIOD. HURRY SUPPLY WON'T LAST LONG AT THIS PRICE

WE HAVE A NEW BBS! PLEASE GIVE IT A CALL AT 415-261-4513 AND LET ME KNOW WHAT .... YOU THINK. BRIAN

Call fo a copy of 15 day trial agreement. Tax & frieght extra. Send check or add 1.90 for COD. Price may change. Store price may differ. While supplies last. No POs, terms, or credit cards

Silicon Valley Surplus
415-261-4506
4401 OAKPORT OAKLAND CA, 94601
OPEN 10am-6pm PST
CLOSED SUN & MON

Reader's

Laser Rangefinder

I am a student in a Science & Technology program, and during our Senior year we must complete a year long research project and Senior Thesis. I am building a portable laser ranging device and I want to hook the experiment up to a ZX81 computer. The laser will stop and start a timer as it is shot and returns from a target. I want to send the time to a program that will calculate the distance to the target.

I was wondering if you have had any prior experience with this particular computer and if you could give me some advice or suggest someone who could. I would be most grateful for any help.

E.T.

Editor's Note: Is anyone willing to offer some help or comments on this project even if it doesn't concern the ZX81?

8-Bit Fan

To date I have found TCJ to be the best publication of many that land on my desk in a week. Always good solid hands on information. Hopefully you will not go the IBM route as did another magazine. We broke our rule with them about more that a one year subscription and after a couple of issues off they went predominantly IBM. Reading items of interest to us in it takes less that five minutes and we are not renewing for that reason.

Somewhere you got our initials shortened up regarding the hard disk problems in issues #25 and #26, the initials should have been JTL not JT. The reply from Tom Hilton in issue 26 was very interesting but not an answer. Certainly we agree with the 8MB limit — BUT our question was how do we add say another 40MB drive to an SB180 system. Our BIOS can go to drive "P". We can only access 32MB of our present 40MB drive now, leaving a lot of unused area. I guess the problem comes down to the formatting program for the hard disk which limits us to four apparent drives, currently E,F,G,H. We are using an Adaptec ACB4000 so have a second SCSI slot available to run another drive.

What appears to be needed are the following:

(1) Modifications to HDINIT.COM allowing us to set up say five drives per 40MB hard disk of say a little under 8MB each.

(2) A way to be able to configure another drive as drives J,K,L,M etc. which doesn't appear possible at present.

Our present hard disk is a NEC D5146. Possibly since it has only 4 data disks item 1 is impossible, but I see no reason that item 2 can not be done.

Unfortunately our programming is done in the higher level languages such as Dbase, Pascal, etc, and the hieroglyphics of HD64180 and Z-80 assembly code are Greek to us. I guess we shall have to take the bull by the horns one day and get down to it.

Thanks for an excellent publication to date. Keep it up and please ensure no defection to the Imbecile's Biggest Mistake.

J.T.L.

Editor's Note: As I understand it, the basic problem is that CP/M can only address 8MB per drive and the hard drive controller can only handle four drives. Thus you can only partition a physical drive into four logical drives of 8MB each, and four times eight equals thirty-two. The answer is to use more than one physical drive and more than one hard drive controller, and partition each physical drive into several logical drives. I also feel safer with my data spread over several physical drives. I would choose 20MB drives, each partitioned into four 5MB sections, and would use ZCPR3 so that all user areas are available.

Feedback

Reader Feedback

I am writing in response to your request for reader feedback.

I feel that TCJ is straying away from my needs. A magazine that fullfills the goals as stated in the Editor's Page in Volume 1, Number 1, is exactly what I need. Why not pick a project, for example your NC controlled milling machine project, and carry it through to a useful end? Where existing hardware is available at affordable hobbiest prices, use it, otherwise build it. Where suitable software is available, use it, otherwise write it.

Don't turn the magazine into a generalized software magazine with too many articles like "Affordable C Compilers" and "Concurrent Multitasking." For that kind of information I subscribe to Dr. Dobbs Journal of Software Tools.

Don't turn into a computer industry "news" magazine with too many articles like "Inside AMPRO Computers" BYTE magaine's subscription gives me that.

While I enjoyed all the above articles, I would rather have read them in magazines that I consider more appropiate for them.

C.D.M.

Editor's Note: Reader's feedback is VERY important, and we encourage every reader to send us a list by mail or on the BBS describing the four subjects that they would like to have covered in future articles.

A Computer Builder

Issue #26 just arrived and I read it cover to cover. Ironically, last week some back issues of TCJ and "The Big One" (you know, the magazine with 300 pages of advertising, been around for ten years) arrived. I just threw away several years worth of old computer magazines, and comparing the two which arrived on the same day, I found what was missing in computer literature. I sincerely hope you can keep up what you have so diligently started.

I have a North Star Z80A Processor Board model ZPBA2, copyright 1977. It is in an IMSAI computer, and has developed a minor glitch. I have schematics on everything else, do you know where I might find the processor board schematics?

You mentioned, I believe, in a previous issue that you would like to have a library of old schematics and instructions for various archaic systems. If this is true, I can offer copies of the Big Board I (I have two working boards to possibly sell), the Tarbell Casette interface board, VDB-A Video Terminal S-100 card, Jade Z-80 (actually Ithica Z-80) and an Ithica Audio board with the working mod's on it, and the Xitex SCT-100 Single Card Video Terminal.

Right now I am in the final stages of construction on a ZAP-80 microcomputer. This is being built from Steve Ciarcia's book "Build Your Own Z-80 Computer" (Byte Books, 1981). You should have this in your offerings, it is of great interest to some, if not many, of your readers. The circuit he uses was the winner in a competition of four different possibilities.

What was needed was a single board controller/computer. It is to serve as a data logger, and whatever else comes to mind. For several years I have been trying to find a "public domain computer" sufficient for a beginning student to wire wrap and use for some significant purpose. There are simpler solutions to the problem (such as the 8748 single chip microcomputer). However, this Z-80 board has some extreme advantages. First, all of the software will be upward compatible (the next step up might be a Little Board®). Second, all lines off the CPU are buffered with the expectation of adding odd-ball devices to the exterior. Third, there are two 8-bit parallel ports (input) and an 8-bit output port, with an optional RS-232 serial port. To make the projects initially useful, it will be an en-

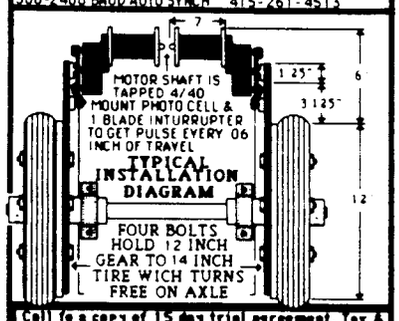
(Continued on page 50)

BUILDING A ROBOT?

MOTION PACKAGE ONLY 99.00
DIRECT GEAR DRIVE FOR WHEELS 12 INCH OR LARGER
INCLUDES:
2ea DC MOTORS
2ea 16:1 GEARBOXES
2ea 24 TOOTH STEEL SPUR GEARS
2ea 60 TOOTH NYLON IDLE GEARS
2ea 235 TOOTH 12 INCH DRIVE GEARS
MOTOR, GEARBOX, & IDLE GEAR ARE PRE-ASSEMBLED. DRIVE GEAR IS READY TO MOUNT ON YOUR WHEEL.

VDC	RPM	NO LOAD	STALL	MPH	MPH WITH 12" WHEEL USE A LARGER WHEEL TO GO FASTER
24	34	800 MA	10.0AMP	125	
18	27	600 MA	7.5AMP	96	SEND FOR
12	18	500 MA	5.0AMP	64	COMPLETE
6	8	450 MA	2.5AMP	21	LISTING

MORE ROBOTICS PARTS AVAILABLE CALL OUR MODEM
300-2400 BAUD AUTO SYNCH 415-261-4513


7
MOTOR SHAFT IS TAPPED 4/40
MOUNT PHOTO CELL & 1 BLADE INTURRUPTER TO GET PULSE EVERY .06 INCH OF TRAVEL
1.25"
6
3.125"
12
TYPICAL INSTALLATION DIAGRAM
FOUR BOLTS HOLD 12 INCH GEAR TO 14 INCH TIRE WICH TURNS FREE ON AXLE

Call fo a copy of 15 day trial agreement. Tax & frieght extra. Send check or add 1.90 for COD. Price may change. Store price may differ. While supplies last. No POs, terms, or credit cards

Silicon Valley Surplus
415-261-4506
4401 OAKPORT OAKLAND CA, 94601
OPEN 10am-6pm PST
CLOSED SUN & MON

$7.00 EQUIPMENT CASE

SIMULATED LEATHER FINISH 10 INCH BY 12 INCH BY 4 INCH RUGGED ABS PLASTIC WITH CHROME PLATED LATCHES
USE IT FOR A CAMERA CASE, TOOL BOX, OR PORTABLE EQUIPMENT CASE. GREAT WAY TO CARRY THOSE BOOKS AND SOFTWARE TO AND FROM WORK

$29.00 PRECISION X Y TABLE

TWO GLOBE DC GEARMOTORS.
LINEAR POT FEEDBACK
USED IN MICRO FITCH READER
TABLE HAS 4.5" BY 6" TRAVEL

DC GEARMOTER $14.00
5 VOLT TO 12 VOLT DC OPERATION
250 TO 1 GEAR REDUCTION.
A PAIR OF THESE WILL MOVE A 500 LB ROBOT

64K DRAMS .75 EACH
$6.00 FOR A PKG OF 8
SOLD IN PKG OF 8 OR 9 ONLY
NEW CHIPS TESTED BY US AND EACH ONE GUARANTIED TO WORK
.83 EA IF YOU BUY 8 OR LESS

IC SALE THIS MONTH ONLY

HM6116	.40	82C51	1.99
74LS393	.60	6502	1.99
74LS245	.50	FDC 765	2.99
4517	.40	80C86	3.99

CALL OUR MODEM CATALOG
415-261-4513

Call fo a copy of 15 day trial agreement. Tax & frieght extra. Send check or add 1.90 for COD. Price may change. Store price may differ. While supplies last. No POs, terms, or credit cards

Silicon Valley Surplus
415-261-4506
4401 OAKPORT OAKLAND CA, 94601
OPEN 10am-6pm PST
CLOSED SUN & MON

A Tutor Program for Forth

Writing a Forth Tutor in Forth

by Bill Kibler

Over the past few months I have been working on tutor programs and produced one for a computer class using BASIC. The program just presented menus and then checked for keyboard input. Next it jumped to a given line and ran the code. The code in this case was text statements and took forever to enter. As far as program statements I used both "ON X GOTO" and "IF..THEN" statements. My overall feelings about the program was that it involved too much work for too little output. Needing to do another tutor program and also wanting to do something in Forth, I decided to try a tutor program using F83.

F83 is the most current version of the FORTH 83 standard and is available both commercially and in public domain. F83 has several extra features and is quite an improvement over previous versions of Forth. People who have used other versions of Forth find F83 to be a real challenge to learn (same words new meanings). People who have not used Forth previously are usually lost immediately after the prompt. I figured I needed a tutor and help program that would be available after the prompt. After I started writing the program I was amazed at how easy and how fast I was able to write the mechanics of the program.

I spent about two days thinking about how I wanted information moved and displayed, and then about one hour generating screens to do it. I only had one time in which any code went south for the winter, other than that it was very painless. Although this program is rather simple, involving defining barely ten words, it has made a real convert of me. Compared to the tutor program I wrote in BASIC, this was a breeze to write and will work many times better.

TUTOR.BLK

The program has three variables which keep track of the current screen. Forth stores all information in screens of 1024 characters each. This gives a dislay

```
SCR #0          TUTOR.BLK

(INTRO TEXT FOR SCREEN ZERO                                    BDK112186)
*****************************************************************
*****************************************************************
*****************************************************************
*****                                                         ******
*****          F83 TUTOR AND HELP PROGRAM                     ******
*****          F83 TUTOR AND HELP PROGRAM                     ******
*****                                                         ******
*****                                                         ******
*****            Written   by   Bill Kibler                   ******
*****         PO BOX  487  Cedarville, CA 96104               ******
*****                                                         ******
*****          ALL Commercial rights reserved                 ******
*****                                                         ******
*****************************************************************
*****************************************************************

SCR #1                TUTOR.BLK

( LOAD BLOCK AND START OF TUTOR PROGRAM                        bdk120186)
( variables and display  routines  )
VARIABLE ETUTOR    ( END DISPLAYING  TUTOR SCREENS )
VARIABLE STUTOR    ( BEGINING SCREEN OF CURRENT GROUP )
VARIABLE NTUTOR    ( NEXT TUTOR SCREEN OF GROUP  )

: L$$K    DUP 36 = IF  1 ETUTOR !   THEN  ; ( CHECK FOR $$ )

: DISPLAY      ( DISPLAY SCREEN OF TEXT )
       1 ?ENOUGH  L/SCR 1
            DO          5 SPACES
         DUP  BLOCK I C/L * + C/L
    TUCK PAD SWAP CMOVE PAD SWAP    ( >TYPE WITHOUT THE TYPE )
   0 ?DO   DUP C@  L$$K EMIT 1+ LOOP DROP  ( TYPE WITH L$$K )
   CR       KEY?  ?LEAVE       LOOP DROP  ;
     -->

SCR #2                TUTOR.BLK

( go get screens of information - gotutor tutor        bdk120186)
: WTPRT  20 SPACES ." USE SPACE BAR FOR NEXT SCREEN "  ;
: ESCCHK DUP 27 =  IF 1 ETUTOR ! 32  THEN ;  (  SET ESC FLAG )
: WAIT      WTPRT  13 EMIT  ( PRINT THEN CR WITHOUT LF )
        BEGIN KEY ESCCHK  32 = UNTIL ; ( LOOP TIL SPACE KEY  )

: GOTUTOR       ( DISPLAYS SCREEN ON STACK THEN WAITS  )
         CR DUP SCR !  15   SPACES   .SCR CR     .
         BEGIN  DISPLAY  WAIT  NTUTOR  @ 1 + DUP
         DUP  NTUTOR ! 1  ETUTOR  @ = UNTIL CR CR  15 SPACES
   ." REPT = REPEAT LAST LESSON ...GET = NEXT LESSON  " CR CR ;

: TUTOR      ( STORE SCREEN POINTERS THEN GOTUTOR    )
          0  ETUTOR   !
         DUP DUP STUTOR ! NTUTOR ! GOTUTOR ;
      -->

SCR #3                TUTOR.BLK

( INITIALIZE AND START THE LOOPS..GET..REPT..                  BDK112586)

: GET       ( GO GET NEXT GROUP OF SCREENS    )
      NTUTOR  @        TUTOR    ;
```

```
: REPT      (  GO BACK AND REPEAT SET OF SCREENS   )
      STUTOR @  TUTOR   ;

: START-TUTOR    (  START WITH FIRST SCREEN OF TUTOR   )
           10 TUTOR   ;

: HELP          (  GIVE INTRO MESSAGE    )
          6 TUTOR  ;

SCR #4                  TUTOR.BLK

( DEFINING MODULES OF INFORMATION.....                    BDK112186)

: CHP1            10 TUTOR  ;
: CHP2            20 TUTOR  ;
: CHP3            30 TUTOR  ;
: CHP4            40 TUTOR  ;
: CHP5            50 TUTOR  ;
: CHP6            60 TUTOR  ;
: CHP7            70 TUTOR  ;
: CHP8            80 TUTOR  ;
: CHP9            90 TUTOR  ;
: CHP10          100 TUTOR  ;
: CHP11          110 TUTOR  ;
: CHP12          120 TUTOR  ;

   -->

SCR #5                  TUTOR.BLK

( MORE ROOM FOR LESSON WORDS....                          bdk120186)

: GLOS            130  TUTOR ;




: PRTSCR              CR ." CURRENT GET SCREEN IS " NTUTOR @  .
          CR  ." REPT SCREEN OF INFORMATION IS " STUTOR @ . CR  ;

 HELP

SCR #6                  TUTOR.BLK

(  PRINT SCREENS FOR TUTOR INFORMATION...                 bdk120186)

          FORTH-83 TUTOR PROGRAM AND HELP SCREENS
               WRITTEN  BY   BILL KIBLER
                     NOV/DEC 1986
               ALL COMMERCIAL RIGHTS RESERVED


   This program is intend to help beginners and old FORTH users
alike. The screens contain information on FORTH-83 and are
related to the recomended book " STARTING FORTH "  by Leo Brodie
which can be used as a textbook with this program. Each chapter
or series of screens is organized to present the words used in
the chapter first in a glossary form. Old timmers will find this
glossary important to see the differences between F83 and other
versions. Typing HELP will repeat these screens, then typing

SCR #7                  TUTOR.BLK

( second intro screen with list of words...               bdk120186)
the chapter number for the area of help needed. Typing ESC key
will exit the screens and return to the system prompt.  GET will
display next chapter of  information, while REPT will start
with the first screen of the chapter again. START-TUTOR will
start with the first chapter. Display will pause  until
SPACE BAR is pressed.....   NEW F83 WORDS.....
```

of 60 characters by 16 rows, which at first I felt too small for the tutor program. After generating some text I found the size actually to be a very nice presentation. The borders and display should be usable on any machine, one of Forth's strong points. The other words merely set which screens to load and where you started. The only major word I defined was actually modifiying the Forth LIST word. Normally using LIST will display a screen with each line numbered, and with line 0 containing the origination date and the title. I wanted the starting title and no line 0 or line numbers. What I did was reproduce the original F83 LIST definition and then remove or rearrange the information to get the desired results. I made one mistake during this process which sent the machine into the operating system, but I had saved each screen to disk and only had to reboot to make corrections and try again.

The first word you can use is HELP, which gives you an introductory set of screens to the program. This introduction gives you some of F83's words both in the tutor program and regular F83. The "**$$**" means the end of a series of screens and will drop the program back into F83 as indicated by the "OK" prompt. "REPT" will repeat the series of screens again, while "GET" will start the next set of screens. Eliminating the line 0 and some extra returns enabled the next screen's text to display immediately after the first (after you hit a space bar). "ESC" will terminate the displaying of screens, but "GET" will start where you left off. "PRTSCR" will tell you what each of the two variables contains (t screen pointers GET & REPT).

To enter text, you use the built in F editor and start each screen with NEW". This will allow you to enter te one line after the other, while editi previous lines will require you looking the commands until you get used them. Now that I am getting used to the commands I have found the editor to be OK, especially when you consider it is available at all times. I am planning on writing the tutor screens along the lines of Leo Brodies "Starting Forth", but felt the idea of getting tutor programs out there more important than the text itself.

Conclusion.

In using other tutor programs I have found them both difficult to use and impossible to fit my needs. I found most users either know too little or too much to make proper use of the program's infor-

```
The following words are important utilities in F83 and may be
different from previous versions. WORDS will display a list of
F83 words used. OPEN allows use of an existing file, EXTEND is
used to add screens, and CREATE is used for making a new file.
INDEX displays a list of line 0, use 1 20 INDEX to list
screens 1 to 20. 1 30 SHOW will print 6 screens to a page on
your printer in condensed mode ( use ' EPSON IS INIT-PR for
epson printers). 1 30 TRIAD prints three to a page if condensed
print is not available. 1 30 SHADOW SHOW  will print both the

SCR #5                  TUTOR.BLK

( THIRD  PRINT SCREEN OF TUTOR INFORMATION.....        bdk120186)
regular screens on the left side and the information screen on
the right side of each page. SEE xxxx disassembles the word
xxxx, while VIEW will open the source file ( on A: drive) and
list the screen it is in. VOCS will list the vocabularies in
the dictionary, while ORDER displays the path of the directory
search. Use DOS WORDS to see a list of the DOS dictionary words.
CAPACITY will print the number of screens in a open file. A L
will toggle between the shadow and the source screens. N L will
display the next screen, L will list current screen, B L will
list previous screen. 1 EDIT will invoke the line editor with
screen 1 ready to edit. 0 NEW will start editing at line 0 and
and allow the text to be entered one line after the other. HEX
100 80 DUMP will do a hex dump of memory location 100h to 180h.
DEBUG LIST will allow stepping through list when used next as
in 1 LIST.

SCR #9                  TUTOR.BLK

( last   intro screen with list of words...            bdk120186)

                        TUTOR WORDS
  CHP1 = introduction          CHP2 = fundamentals
  CHP3 = RPN and STACK         CHP4 = editor commands
  CHP5 = conditionals, nests   CHP6 = fixed point operations
  CHP7 = loops  ( & nested)    CHP8 = number types
  CHP9 = var. const. arrays    CHP10= F83 structure
  CHP11= Input/Output          CHP12= extensions
  GLOS = alphabetical glossary of F83 words

  GET = next chapter           REPT = begin chapter again
  HELP = repeat these screens  START-TUTOR = start at CHP1
  SPACE BAR = next screen      ESC = stops display
  PRTSCR = GET and REPT pointers
                                                        $$
```

mation. A typical problem is getting into and out of lessons. Many times you can get out but must start all over with lesson one again. A recent tutor program I must use makes you enter information at each prompt before you can continue on. This is not acceptable as the only information I want is at the end of each lesson (a review and glossary). My Forth tutor program gives you many options, from printing an index (or the whole thing) to LISTing a single screen. After each lesson you are given the Forth "OK" prompt and can enter any F83 word, try new words, run samples from the lesson, or leave F83 by going "BYE".

I found using Forth to be easy and simple. I use many different systems and CPUs which started me on Forth and will be the only language I could use on all the units. Currently I have to learn everything all over again when ever I change systems, while learning Forth would allow me to learn only one system. F83 contains most functions you might consider when developing a program or changing a system, and as such could be the solution to efficiently using many systems. Give it a try. ■

Editor

(Continued from page 3)

make the minor revisions so that a program does what we want it to do. My first choice is a program which includes the source, and my second choice is a program which I can figure out how to patch.

Clark Calkins' method of disassembling programs in order to provide modifiable source code is very important for those of us who are working with CP/M programs which are not supported because of the manufacturer's neglect or the fact that they are out of business, and we are very thankful that Clark is taking the time to share his knowledge with us in this first article of a series. I would like to see source code generators for popular, but undocumented, CBIOSs, disk format programs, boot PROMs, etc. If you have a likely candidate and would be willing to cooperate in a group effort (we can't expect Clark to do all this work for us), drop me a note or post a message on our BBS. If there is enough interest we'll organize a SIG.

While talking about source code, note that Hawthorne's 68000 operating system includes the source code plus the compilier for the language in which it is written.

8-bit Developments

There is still a lot of activity in the 8-bit domain with Z-80 and HD64180 single board computers selling well. There are also a lot of 8-bitters still in use and they are ideal tools for learning about interfacing and how operating systems work.

Toshiba and Zilog should both know how many 8-bit chips are being sold, and they have just announced new Z-80 family devices which means that they plan on producing a lot more of them.

Toshiba has introduced a line of Application Specific Standard Parts (ASSPs) which include a CMOS Z-80 CPU with peripheral functions on one chip. Their Z84C011AF which combines the Z-80 with a clock generator controller, a counter timer, and five 8-bit parallel I/O ports comes in a 100 pin flat pack in 6 or 8 MHz with 10 MHz promised. The I/O ports are bit enabled and each port bit can be used as input or output. The Z84C013AT is designed for dedicated communications with serial I/O, and the Z84C015AT is designed for communications with parallel I/O.

Zilog has announced their Z84C80 CMOS chip which can replace 70 to 100 SSI/MSI chips in a Z-80 system, and in-

cludes a clock oscillator, watchdog timer, Z8500 bus interface, dynamic RAM interface that handles 64K and 256K memory chips, and decodes the instruction set which permits code and data to be separated into different address spaces and doubles memory space.

BD Software is now shipping revision 1.6 of their C compiler. This revision includes the RED editor (with source code) which reacts interactively with the compiler for easy error corrections, the buffered I/O functions have been changed so that they are now K&R compatible (programs written for V1.5 will have to be modified), the compiler error detection and diagnosis have been improved, plus there are numerous other improvements. Note that the source code for all the library functions is included. The BDS C compiler has been my first choice as a programming language for writing text filters and drivers for our phototypesetter, and the new version is even better.

Another entry is Borland's 8-bit Turbo Modula-2 compiler which is being distributed by Echelon and Micromint, and will be very important for people writing non-trivial programs but who don't like C. Turbo Modula-2 has a lot of the familiar syntax of Turbo Pascal, plus separate compilation and library facilities similar to C. Interfacing to assembly routines has also been greatly improved compared to Turbo Pascal, as Turbo Modula-2 has a CODE statement which reads in an assembly language .COM file during compilation. The assembly routine must be relocatable (only relative jumps) and its length must be included in the first two byte, and this is all spelled out in the manual. It also includes type and version checking across compilation units, and can be used for large program development and for operating system concepts like concurrency and interrupt handling.

I feel that while a lot of work is still being done in BASIC, it should be replaced by Turbo Pascal for trivial programs. I also feel that Turbo Pascal should be replaced by C or Trubo Modula-2 plus assembler for non-trivial programs. I didn't find anything about producing ROMable code in the Turbo Modula-2 manual, so BDS C may have an advantage there.

TCJ intends to support both BDS C and Turbo Modula-2, and is especially interested in acquiring extended library modules to be published in the magazine and posted on the BBS. Send us information on what you are doing, and share your developments with others.

FREE CATALOG AND SOFTWARE APPLICATIONS GUIDE

CP/M MSDOS

AFFORDABLE ENGINEERING SOFTWARE

TRSDOS PCDOS

ANALYSIS

XFER $72.95

LOCIPRO $72.95

SPP $72.95

STAP $72.95

CIRCUIT DESIGN

ACTFIL $72.95

ACNAP $72.95

DCNAP $72.95

GRAPHICS

PDP $72.95

PCPLOT $72.95

PLOTPRO $72.95

TEKCALC $72.95

MATHEMATICS

TEKCALC $72.95

MATRIX MAGIC $72.95

COMCALC $72.95

REPORT WRITING

Right Writer $97.95

BV Engineering
Professional Software

2200 Business Way, Suite 207
Riverside, CA 92501 USA

(714) 781-0252

Feedback Control

The article on Feedback Control Systems by BV Engineering is a good example of both engineering software and a very important subject for any type of control system.

Feedback control can be applied to numeric machining, robotics, temperature control — and even magazine publishing. To keep TCJ steering down the right path we need to know what the readers want and how we are deviating from the path. We have renamed the the readers letter column "FEEDBACK" to better describe its purpose, which is for the readers to ask questions, respond with answers, talk about what's happening in the computer field; and most importantly, where TCJ should be going. Take time to write or to post a message on our BBS.

Computer Parts

In the previous issue, I mentioned that the power supply burned out in one of our Morrow S-100 systems. I was concerned about finding a replacement since Morrow is out of business, but I located Silicon Valley Surplus (4401 Oakport St., Oakland, CA 94601, phone 415-261-4662) who obtained much of Morrow's parts inventory. A replacement supply was only $25, and in addition I bought a spare DJDMA disk controller board (complete with the latest ROM) for only $45. The people I talked to there were very pleasant, helpful, and knowledgeable. They even have an on-line bulletin board to list some of their latest bargains. See their ads in this issue for some great deals including the Morrow Micro Decision mother board which makes a great single board computer. ■

Forth Fan

Don't let us down! I still wish there were some Forth articles in TCJ, but I'll be content as long as you continue to support CP/M as well as MS-DOS.

I know I'm not alone when I say that having a MS-DOS machine hasn't made me want to trash my CP/M box(es).

G.S.

Editor's Note: We haven't turned down any Forth articles, we just haven't received many. See Kibler's TUTOR article in this issue, and tell potential Forth authors to contact us. ■

Disk Parameters

Modifying the CP/M Disk Parameter Block for Foreign Formats

by C. Thomas Hilton

A number of our readers have asked for more information on tailoring the CP/M® BIOS, and specifically on how to modify the BIOS in order to read other disk formats. Information on writing to the disk is very hardware specific, but there are some generalities and the information on the AMPRO Little Board® will serve as a starting place for those with other systems.

We must keep in mind the basic structure of CP/M, or CP/M derived operating systems. The CCP interfaces with the human operator, and relies upon the FDOS, (Floppy Disk Operating System), segment for all hardware interface. The FDOS segment deals with all hardware requests in a "sentence" type format. It does not however have much knowledge about what the system hardware looks like. The BIOS, (Basic Input/Output System) is the only segment which deals directly with the system hardware. Because the other segments of the CP/M system require no hardware specific information CP/M may be run on nearly any manufacturer's hardware. This is the purpose of the CP/M concept, and why we don't have "compatibility" problems the way the 16 bit systems do.

Less than a fifth of a standard BIOS is concerned with tasks not associated with disk management. When we begin to think of disk management we are concerned with the entire BIOS and cannot simply isolate a given segment and work with it. Far too many concepts, or 'small details' are interrelated, and these 'small details' form the dominate four fifths of the BIOS.

Getting Started

From here on we will refer to a generic operating system as the "DOS", which is assumed to be ZRDOS, CP/M, Hermit DOS, or any other CP/M derived operating system.

The DOS can manage nearly any type of disk storage device we care to connect to the system. These devices may range from a single density 5 inch drive to a quad density, or for the little AMPRO, 80 megabytes of hard drive storage. Wait just a minute! Didn't he just say the DOS segment has little knowledge of the disk hardware? Yes I did. There isn't really a conflict of concepts. In beginning we will have to view the disk system as the DOS sees it, and work our way deeper into the concept as we go. Any other route in our quest would lead to miles upon miles of dull theory. I would rather do than read, wouldn't you?

Of all the thousands of details concerned with disk management only a few are of real concern to the DOS segment. The DOS relies upon the BIOS to take care of these details. The DOS asks the BIOS to provide it with a set of "parameters" describing the disk device. Many times, as we attempt to interface new technologies with standard, and aging systems, (all the better to upgrade them as we are doing here), the BIOS will lie to the DOS segment about the disk drive. What the DOS doesn't know won't hurt it. The most common occurence of misinformation is about hard-drives, which were not common when CP/M was written.

Primary Disk Parameters — Some Theory

The DOS asks the BIOS for parameters which "describe" what the mass storage device looks like. We will cover these parameters several times, with a slightly different perspective each time. We have to begin somewhere, and what's good for the DOS is not always good for the BIOS.

The size of the blocks of disk space used by the concerned format is a trade-off decided upon by the author of the BIOS. It is said that "He who writes the BIOS owns the company." There are many reasons why this is true and we will discover them.

The allocation block may assume only a limited number of sizes to remain compatible with the DOS. Of these limited values at least one is considered obsolete. All possible values are related to other disk parameters. The size of the allocation block, simply expressed for now, is the amount of disk space used in all transactions. A better way of putting it is that if a two character file shows up in the directory as using 2K of disk space then you may assume your system uses a 2K allocation block size. The size of the allocation block is a great headache for the format designer. File space is allocated only in terms of blocks. In the two character file example there is a large waste of disk space. We can expect that at least half of the last block in a given file will be wasted. This waste factor is what prompted the LU.COM and NULU.COM library utilities to be created — they free up much of this wasted space within the individual files.

Now then, it seems logical that the larger the allocation block the greater the amount of space which will be wasted. This is a valid assumption. The considerations are the size of the disk being used, and the number of files likely to be placed upon it. It would follow that the lower the disk capacity and the greater the number of files to be placed upon the disk, the smaller the allocation block should be. It is not that simple, even if we knew for certain what kinds of drives those darned TCJ readers would attempt to connect to our system!

The DOS "sees" the storage device in terms of allocation blocks, which we will refer to only as "blocks" from here on. It knows only how many blocks the disk will hold. The concepts of sectors and tracks it leaves to the BIOS to figure out. The total number of blocks available on the disk allows a "map" of the disk, in allocation blocks, to be built.

If the disk is a small one of less than 257 blocks, the map for the entire disk may be referenced in a single byte. If the disk is a large one we will have to use a word, or two bytes, to map the disk. To avoid ambiguity in map making, a two byte map should hold a predefined set of numbers. The size of the allocation block therefore directly effects the production of a map of the disk. The disk capacity and the block size determine how many block numbers will fit in the data map of the disk. If we make the number of blocks less than 256, a single byte, then the block size must be large, a single directory entry can command more disk space, and we will waste more disk space, (because of the large

block size).

The block size and disk capacity determine the amount of space which can be described by a single directory entry. This amount of disk space controlled by a directory entry is called a "data map." The amount of space controlled by each entry in the data map will be either eight or sixteen times the size of an allocation block. This is a design rule. This number is also represented by a term called a "logical extent" which will always be a multiple of 16,384. This is another design rule. There is a case where these rules could be violated. If the block size were 1K and the disk was so small that there were less than 256 blocks, then a directory entry could command only 8K of disk space. This is not a usable combination of disk parameters. This combination is not allowed in modern design concepts. The use of 1K allocation blocks may only be used with disks of less than 256K of data, such as "fruit-loop" computers and other low-end consumer toys.

The block size also effects the relationship between the logical extent and the physical extent, (the actual amount of space commanded by a single directory entry). The larger the physical extent, the fewer the number of directory entries which will be required to describe a complete file. The size of the physical extent is more than a theoretical design concept as it directly affects system performance. It takes time to locate and "open" a new extent; a seek of the disk head back to the directory track is required. It follows that the larger the physical extent the less time would be required to process a large file. The time it takes to find data on the disk is important when we begin to think about how the system will perform with random access files. But, as we have briefly discussed, the larger the extent, or block size, the more wasted space we have on the disk.

Another consideration about the size of the allocation block is the relationship to the number of directory entries it takes to describe the entire disk. We must consider two extremes of disk space management. On the one hand we must consider that someone may place all his files in a library and the entire disk may be filled with only one file. A single directory entry therefore must be capable of describing the entire disk, if it has to. If the directory were constructed any smaller than that which could command the entire disk space it would not be possible to use the entire disk. On the other hand the disk may be filled with two character files commanding a single block. There must be enough directory entries to handle the load. From what we already know the smaller the block size the more directory entries would be required to describe the entire disk.

Just to make things interesting let's establish the rule that there cannot be more than 16 allocation blocks used by the directory. Yes, you had better dust off that old calculator and send out for another bottle of you favorite headache reliever! The basic rule of thumb here is that the smaller the block size the larger the directory must be. It would also be logical to think that the size of the disk would play a major role in determining the allocation block size since it affects so many "small" considerations.

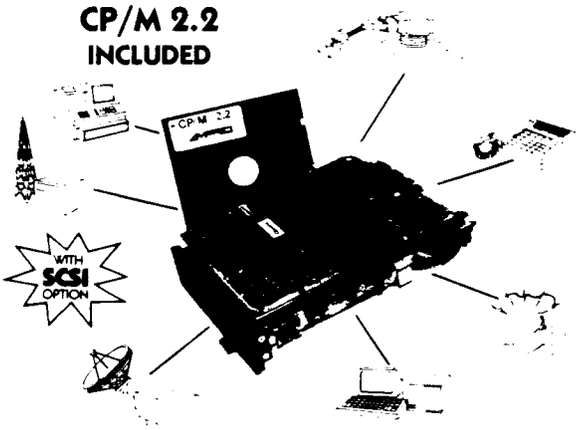
The DOS requires the BIOS to tell it the number of blocks a disk will hold. This figure represents the capacity of the disk. When all the blocks are assigned to a directory entry the disk is full, end of story.

Right from the start I am going to run afoul of CP/M pundits. The parameter I refer to as "Records per Track" is generally called "Sectors per Track." What it refers to is the number of 128 byte data segments per track. Sectors have not been 128 bytes per track since the 8080 processor was new. So,

Little Board™ $249

The World's Least Expensive CP/M Engine

CP/M 2.2 INCLUDED



- 4 MHz Z80A CPU, 64K RAM, Z80A CTC, 4-32K EPROM
- Mini/Micro Floppy Controller (1-4 Drives, Single/Double Density, 1-2 sided 40/80 track)
- 2 RS232C Serial Ports (75-9600 baud & 75-38, 400 baud), 1 Centronics Printer Port
- Power Requirement: +5VDC at .75A; +12VDC at .05A / On board -12V converter
- Only 5.75 x 7.75 inches, mounts directly to a 5-1/4" disk drive
- Comprehensive Software Included:
  - Enhanced CP/M 2.2 operating system with ZCPR3
  - Read/write/format dozens of floppy formats (IBM PC-DOS, KAYPRO, OSBORNE, MORROW ...)
  - Menu-based system customization
  - Operator-friendly MENU shell
- OPTIONS:
  - Source Code
  - TurboDOS
  - ZRDOS
  - Hard disk expansion to 60 megabytes
  - SCSI/PLUS™ multi-master I/O expansion bus
  - Local Area Network
  - STD Bus Adapter

BOOKSHELF™ *Series 100*

Fast, Compact, High Quality, Easy-to-use CP/M System

Priced from $895.00
10MB System Only $1645.00

- Ready-to-use professional CP/M computer system
- Works with any RS232C ASCII terminal (not included)
- Network available
- Compact 7.3 x 6.5 x 10.5 inches, 12.5 pounds, all-metal construction
- Powerful and Versatile:
  - Based on Little Board single-board computer
  - One or two 400 or 800 KB floppy drives
  - 10-MB internal hard disk drive option
- Comprehensive Software Included:
  - Enhanced CP/M operating system with ZCPR3
  - Word processing, spreadsheet, relational database, spelling checker, and data encrypt/decrypt (T/MAKER III™)
  - Operator-friendly shells, Menu, Friendly™
  - Read/write and format dozens of floppy formats (IBM PC-DOS, KAYPRO, OSBORNE, MORROW ...)
  - Menu-based system customization

DISTRIBUTORS

ARGENTINA: FACTORIAL, S.A., (1) 41-0018, TLX 22408 **BELGIUM:** CENTRE ELECTRONIQUE LEMPEREUR, (041) 23-4541, TLX 42621 **CANADA:** DYNACOMP COMPUTER SYSTEMS LTD., (604) 879-7737 **ENGLAND:** QUANT SYSTEMS, (01) 253-8423, TLX 946240 REF:19003131 **FRANCE:** EGAL+, (1) 509-1800, TLX 620893 **SPAIN:** XENIOS INFORMATICA, 593-0822, TLX 50364 **AUSTRALIA:** ASP MICROCOMPUTERS, (613) 500-0628 **BRAZIL:** CNC-DATA LEADER LTDA., (41) 262-2262, TLX 041-6364 **DENMARK:** DANBIT, (03) 66-90-80, TLX 43558 **FINLAND:** SYMMETRIC OY, (0) 585-322, TLX 121394 **ISRAEL:** ALPHA TERMINALS, LTD., (3) 49-16-95, TLX 341667 **SWEDEN:** AB AKTA, (08) 54-90-90, TLX 13702 **USA:** CONTACT AMPRO COMPUTERS INC., TEL.: (415) 962-0230 TELEX: 4940302

IBM®, IBM Corp., Z80A®, Zilog, Inc., CP/M®, Digital Research, ZCPR3™ & ZRDOS™, Echelon, Inc., Turbo DOS®, Software 2000, Inc., T/MAKER III™, T/Maker Co.

AMPRO COMPUTERS INCORPORATED

67 East Evelyn Ave. • Mountain View, CA 94041 • (415) 962-0230 • TELEX 4940302

we of the enlightened "New Wave of Eight Bit System Designs" will tell it like it is and refer to this parameter as records per track. Each RECORD we will determine to consist of 128 bytes of user data.

The DOS doesn't know anything about sectors and tracks, nor their size. This is the realm of the BIOS where only those with at least a brown wizard's robe with gold stars might travel. However, the BIOS demands to speak in the terminology of sectors and tracks so the DOS must fake it. Knowing the number of standard RECORDS per track it can do some fancy math calculations and tell the BIOS what it needs to know to return a specific record from the disk's surface. This is not as easy as one would suspect as records are laid down anywhere there is space, seldom one after the other.

The Disk Parameter Block

In actual application the parameters we have just discussed are returned to us in a fifteen byte data structure called the "Disk Parameter Block," which we will refer to as the "DPB" from here on. The address of the DPB is returned from the BIOS using DOS function call #31. The structure of the DPB is shown in Figure One.

BYTE	NAME	OLD MEANING	NEW MEANING
XX	SPT	SECTORS PER TRACK	RECORDS PER TRACK
X	BSH	BLOCK SHIFT FACTOR	
X	BLM	BLOCK MASK	
X	EXM	EXTENT MASK	
XX	DSM	DISK SPACE MAXIMUM	DISK BLOCK MAXIMUM
XX	DRM	DIRECTORY MAXIMUM	DIRECTORY NUMBER+1
XX	ALO/AL1	INITIAL ALLOCATION	ALV INITIAL ALV MAP
XX	CKS	CHECK AREA SIZE	
XX	OFF	TRACK OFFSET	RESERVED TRACKS

Figure One: The Standard Disk Parameter Block

The first byte in the DPB is the RECORDS per track parameter value. Under CP/M 1.4 there were 128 bytes per sector only. This is why CP/M 1.4 is no longer in use. Modern technology allows several different values of SECTORS per track, with 128 bytes per record within a given sector. This value represents the number of RECORDS per track, and has nothing to do with the number of physical sectors per track. This is where the distinction lies, and must be recognized if we are to battle confusion. If you think in terms of sectors, with every sector being 128 bytes, then nothing will make sense from here on. Think in terms of 128 byte RECORDS only!

The next entry in the DPB refers to the BLOCK SHIFT FACTOR. Recall I mentioned the DOS doesn't know anything about sectors and tracks. This is the first of several data which allow the DOS to calculate where a given record is upon the disk, and demand it of the BIOS. The block shift value is the number of times a given record's number should be logically shifted to the right in a 16 bit register to obtain its allocation block number. The size of the allocation block may be determined by initializing a 16 bit register with 128 and shifting this value LEFT the number of times in the BSH byte.

The BSH is followed by the BLOCK MASK byte which contains a value which, if logically ANDed with a record number will produce the RELATIVE NUMBER of the record within its allocation block.

The EXTENT MASK field of the DPB shows the relationship between the physical and the logical extent. The logical extent is derived from dividing the record number by 128. The physical extent may also be determined from the extent mask, but it is generally easier to get the record number, shift it right BSH times, and then divide the result by eight or sixteen, depending upon the number of blocks in the data map. The DSM, or DISK BLOCK MAXIMUM word reveals the highest possible allocation block number available for the disk of concern. Remember that in computers the number zero is the first number, and a valid number. Hence the number of allocation blocks, or the highest allocation block number, is one more than the value in the DSM. The size of the disk may be computed by initializing a 16 bit register with the value of the DSM+1, and double that value BSH-3 times. The number of records may be determined by doubling the result BSH times. If you are slick, as the byte count will overflow a 16 bit register, the disk size in bytes may be determined by DSM+1 doubled BSH+7 times.

The maximum number of directory entries, the DIRECTORY NUMBER+1, contains the maximum number of directory entries, plus one, contained upon the concerned disk. This is a word sized parameter, and indicates that the DOS is able to support up to 256 directory entries. This large number of directory entries is generally found only in hard disk systems.

The ALV, or INITIAL ALLOCATION VECTOR is a word sized parameter which the DOS maintains for each "active" disk. This word is a string of DSM+1 bits in length in which each bit refers to an allocation block. If the bit is "high," a logical "1," then the block is in use. Each byte in the initial ALV may be used to build a full sized allocation vector for the concerned disk, with these bytes becoming the "leader bytes" for the vector. Some number of the ALV bits will be set high to permanently reserve the directory area of the disk.

Next on the list is the CKS word, which is the DIRECTORY CHECK SIZE parameter. This word sized parameter is used by the DOS to see if the disk on the concerned drive has been changed by the user since the last "warm boot." If the disk is determined to have been changed it is assigned a "read only" status and we are given the all too familiar error message when we try to write to it. If the CKS word is a zero value this "disk has been changed" testing will not be done. This is the case only in hard disk systems, or where the medium is assumed to be "nonremovable." If the CKS value is not zero this word will be given a value representing the number of records in the disk's directory, or DRM+1/4. Directory entries are packed four to a record.

The final entry in the DPB is the OFF word, or RESERVED TRACKS OFFSET value of the concerned disk. This word tells the DOS how many tracks are reserved for the operating system. Knowing that some tracks are reserved the DOS will add this offset to the track number it computes using the record number and the RECORDS PER TRACK parameters of the DPB. It may be interesting to note that the system tracks are always the first tracks on the disk, hence the OFFset value points to the first byte of the directory area on the disk. The more enlightened reader may wonder, at this juncture, why this is a two byte field when only one or two tracks are normally reserved for system use. Well kind reader, the DOS is prepared to handle 65K of reserved tracks, this is true. The reason this is done is to allow the use of hard disk systems. If the medium is continuous, of say 20 megabytes capacity, and CP/M allows a maximum of eight megabytes per disk, how does one account for the palm sized AMPRO's ability to support 88 megabytes of disk storage? You are correct if you feel the offset value is used to "lie" to the DOS segment. On the 20 megabyte Little Board I use for writing it is not unusual to see TWO THOUSAND

TRACKS reserved on LOGICAL DRIVE D! This offset value is used to partition a large hard disk into more bite sized portions compatible with the various limitations of disk space management. Adding 2000 tracks to a record number points to the directory tracks on logical drive D of the hard disk. This is a good way to find your way around the void of a large disk system! The DOS assumes there are four physical disk drives, A, B, C, and D, connected to the system. The BIOS knows there is only one physical drive. What the DOS does not know will not hurt it.

The Reality of the Situation

A full description of how to apply the variables we have just discussed would be beyond the scope of what we wish to achieve, and the design of each reader's disk would be different. Each would make different trade-offs according to his needs, opinions, phase of the moon, and other factors unmentionable in print. The results of these decisions, the values placed in the DPB, and the many algorithms which would be required to support them would be called "DISK FORMATS," and each of them would be incompatible with another's. Are you beginning to get the picture on why there are so many different disk formats, and why no one wishes to second guess the thoughts of the author of a BIOS?

The BIOS is the only portion of the operating system which speaks directly to the system hardware, and the only program, user or otherwise, which should speak to the system hardware. It would follow therefore that each BIOS is concerned only with the specific disk controller chip chosen by the hardware designer. It is also proper to assume, if you wished to physically format a disk in any style other than that specified by the author of the BIOS, you will be required to write your own algorithms to deal with the hardware specific disk controller chip.

The AMPRO Emulator Disk Parameter Block

Turning your attention to LISTING ONE you will see several DPB's such as we have been discussing. Three of these segments describe the decisions made for disk format design by the original author of the AMPRO BIOS. These decisions mark the characteristic AMPRO format from all other combinations of disk format design. FIGURE TWO shows the standard configurations of format design practice. We have far to go in a short space. I will leave the study of the various examples for your leisure time.

Let us pay particular attention to the "E" disk parameter block. It may be located by the label, "EPRAM." Before I converted to AMPRO systems I owned a Kaypro 4-84. The "E" disk in the example is set to the Kaypro 4/10 format by default. I traded in the Kaypro for a Televideo terminal. With most of my software library in the Kaypro format, the default setting allows direct access to my software.

Just above the "E" disk's DPB we will find a label entitled, "TYPETAB," which means the disk TYPE TABLE. As if all the different possible parameter values weren't enough to deal with, we now have to deal with the physical characteristics of the disk drive, and a few "small details" about how the alien format designer reads, and writes, his disks. In the default type table make note of the byte containing the value, "0E6H." The value E6 describes the physical Kaypro disk format.

Each bit in the disk type table has a meaning as depicted in the listing's chart. Bit seven defines whether the disk is single, or double density.

Bit six defines the number of sides the disk has, which also defines the number of heads the drive will have.

Bit five defines how the sector numbers are organized. Some formats consider both sides of the disk as a single surface. Sector numbers are numbered with the first sector on side one of the disk, continuing on the other side. Other formats consider each side of the disk as different with sector numbers defined as "side one, sector one/ side two, sector one."

Bit four defines the "track count," which refers to the way the disk is read. In some formats the disks are read down one side of the disk and back up the other side. The alternative is to read the disk down one side, return, switch side, and read down the disk again.

The bit structure this far contains some obvious, and some not so obvious information. That information which is not obvious will seldom be found in any of the target system's technical data. It is one of many circumstances which make dealing with alien disk formats frustrating.

Bits two and three define the size of the disk's allocation block. The "E" disk supports four options of allocation size, 1K, 2K, 4K, and 8K. Other allocation sizes will require change of disk handling algorithms.

Bits one and zero define the size of the sectors. Remember, we are not speaking of 128 byte units! Those are records. We are referring to the physical size of the disk sector.

An examination of the "E" disk parameter block will show that it is a standard DPB. Following the DPB is a group of constants which are called the "skew translation table."

This far our theory has been a logical one, with "many small details" inferring that everything on the disk is in a neat order. Think again, nothing comes this easy. If it did we would be flooded with articles like these. The term, "skew" means that consecutive groups of data are not placed on consecutive sectors within a disk track. Data which should logically follow one another are separated by some predefined number of sectors. Dropping a record every few sectors improves disk performance. Disk access is already slow, and all tricks possible are used to improve access times. The philosophy of the skew is that it takes time for the processor to deal with data. This time factor is such that the processor may not be fast enough to lay down records as fast as the destination spot spins under the heads of the drive. By spacing the sectors the processor "access window" is made larger. The access window is designed to eliminate an additional revolution of the disk for a missed sector. In hard disk systems skew is called the "interleave" factor.

The DOS can determine if skew is used by calling the BIOS function SELDSK, select disk, which returns the address of the DISK PARAMETER HEADER. We will refer to the disk parameter header as DPH from here on. DOS will check the first word of the DPH to see if it is a zero value. If the value is zero then no skew is used, and the record position calculated from the record number represents the actual record position on the disk. If the first word of the DPH is not a zero value it will be the address of the skew table in the BIOS.

The algorithms which deal with the skew factor are often as unique as the results of the other trade-off decisions made by the author of the BIOS. If the disk uses skew the DOS will call the BIOS function SECTRAN to translate the logical record position into the physical record position. This table is required to manage the skew factor of an alien disk, and that it is data we must have to read, or write, an alien format.

We now at least know something of the parameters contained in the DPB, and have a basic understanding of the data we must provide the "E" disk support algorithms. Fine, but where do we get this information?

Figure Two: Reasonable trade-offs of disk format design parameters.

BLOCK SIZE	BLOCK SHIFT	BLOCK MASK	BLOCKS PER DISK	BLOCKS PER DATA MAP	EXTENT SIZE	LOGICAL EXTENTS	EXTENT MASK
1024	03H	07H	<256	16	16K	1	00H
2048	04H	0FH	<256	16	32K	2	01H
			>255	8	16K	1	00H
4096	05H	1FH	<256	16	64K	4	03H
			>255	8	32K	2	01H
8192	06H	3FH	<256	16	128K	8	07H
			>255	8	64K	4	03H
16384	07H	7FH	<256	16	256K	16	0FH
			>255	8	128K	8	07H

NOTE: A 1K BLOCK SIZE WITH MORE THAN 255 BLOCKS PER DISK IS AN ILLEGAL OPTION

Listing One: AMPRO Standard and Emulator Disk Parameter Blocks

```
;
        ORG     BIOS+80H
DPBASE: DEFW    XLT0,0,0,0
        DEFW    DIRBUF,DPARM,CSV0,ALV0
        DEFW    XLT0,0,0,0
        DEFW    DIRBUF,DPARM,CSV1,ALV1
        DEFW    XLT0,0,0,0
        DEFW    DIRBUF,DPARM,CSV2,ALV2
        DEFW    XLT0,0,0,0
        DEFW    DIRBUF,DPARM,CSV3,ALV3
        DEFW    XLT1,0,0,0      ;the e disk
        DEFW    DIRBUF,EPARM,CSV4,ALV4
;
SPARM:                          ;a single-sided ampro disk
        DEFW    40              ;sec/trk
        DEFB    4               ;block shift
        DEFB    15              ;block mask
        DEFB    1               ;extent mask
        DEFW    94              ;disk size -1
        DEFW    63              ;directory max
        DEFB    128             ;alloc 0
        DEFB    0               ;alloc 1
        DEFW    16              ;check size
        DEFW    2               ;offset
;
DPARM:                          ;a double-sided ampro disk
        DEFW    40              ;sec/trk
        DEFB    4               ;block shift
        DEFB    15              ;block mask
        DEFB    1               ;extent mask
        DEFW    194             ;disk size -1
        DEFW    127             ;directory max
        DEFB    192             ;alloc 0
        DEFB    0               ;alloc 1
        DEFW    32              ;check size
        DEFW    2               ;offset
;
DPARM96:                        ;96 tpi double sided
        DEFW    40              ;sec/trk
        DEFB    4               ;block shift
        DEFB    15              ;block mask
        DEFB    0               ;extent mask
        DEFW    394             ;disk size -1
        DEFW    255             ;directory max
        DEFB    240             ;alloc 0
        DEFB    0               ;alloc 1
        DEFW    64              ;check size
        DEFW    2               ;offset
XLT0:                           ;48 tpi skew table
        DEFB    1,2,3,4,5,6,7,8,9,10
;
;       a call to bios function getedisk returns the address of eparm. the type
;table entry is returned as the value returned of the eparm -1. the translation
;table location is defined as the returned value+15. the following definitions
;set the emulator disk as a kaypro 4/10, kaypro 84 series, as the default
;system to be emulated. drive b is the default emulator drive. these values may
;be altered using the ampro multidsk.com utility.
;
;                       Bit: 76543210
;         Density          x          0=single     1=double
;         Sides             x         0=single     1=double
;         Sector #'s         x        0=same       1=continuous
;         Track count         x       0=down       1=down front, up back
;         Alloc unit           xx     00=1K   01=2K   10=4K   11=8K
;         Sector size            xx   00=128  01=256  10=512  11=1024
;                       Bit: 76543210
;
TYPETAB:
        DEFS    4
        DEFB    0E6H            ;disk type byte  (kaypro 4-84 default)
EPARM:  DEFW    28H             ;sec/trk
        DEFB    4               ;block shift
        DEFB    0FH             ;block mask
        DEFB    1               ;extent mask
        DEFW    0C4H            ;disk size -1
        DEFW    03FH            ;directory max -1
        DEFB    0C0H            ;alloc 0
        DEFB    0               ;alloc 1
        DEFW    16              ;check size
        DEFW    1               ;offset
EDSD:   DEFB    1               ;default drive (drive b default value)
                                ;translate table for emulator drive
XLT1:   DEFB    0,1,2,3,4,5,6,7,8,9
        DEFS    10
;
```

Listing Two: Tool To Determine Disk Parameters.

```
; DISKPRAM.ASM
;
;
; DISKPRAM intended use is to determine the the values of the DISK PARAMETER
; BLOCK and the SKEW TRANSLATION TABLE of a system alien to Ampro MULTIDSK
; program.  After the host system parameters are are determined ESET may be
; used to set up the E: drive on the AMPRO system
;
; DISKPRAM displays the disk parameter block and the skew translation table
; maintained in the target bios in any selected system
;
```

```
; DISKPRAM is the combination of DPB.ASM and SKEW.ASM by
; Robert C. Kuhman, Sysop of the Cro's Nest RCP/M.
;
; Revisions:
;               V1.00 as of 27/10/84  Fred Willink
;               V1.01 as of 07/08/86  C. Thomas Hilton
;                                     Converted code to HMAC format
;
BOOT    EQU     0               ;
BDOS    EQU     05H             ;
FCB1    EQU     5CH             ;
;
;  other equates
;
ACROSS  EQU     8
;
; ampro bios disk id save area
;
; ascii equates
;
BELL    EQU     07H             ; bell
TAB     EQU     09H             ; horz tab
LF      EQU     0AH             ; line feed
CR      EQU     0DH             ; carriage return
SPC     EQU     20H             ; space
;
        ORG     100H
;
START:  LD      HL,0
        ADD     HL,SP           ; hl=ccp stack pointer
        LD      SP,STACK        ; set up local stack
        PUSH    HL              ; save the ccp pointer for return
TEST:   LD      A,(FCB1+1)
        CP      '/'
        JP      Z,HELP          ; help
        CP      '?'
        JP      Z,HELP
        CALL    DPB
        CALL    SKEW
        JP      EXIT
;
HELP:   LD      HL,HLPMSG       ; point to help message
        CALL    COSTR           ; display help
;
; exit no warm boot
;
EXIT:   POP     HL              ; recover ccp's stack ptr
        LD      SP,HL
        RET
;
DPB:    LD      A,(FCB1)        ; drive code given ?
        DEC     A               ; convert a=1 to 0
        JP      P,DPB2
        CALL    FCN25
DPB2:   LD      E,A             ; save for select
        ADD     A,'A'           ; make drive code printable
        LD      (DRIVE),A
        CALL    FCN14
        CALL    FCN31
        PUSH    HL              ; save it for dump
;
        LD      DE,SPT          ; make dph values printable
        CALL    CVT2
        LD      DE,BSH
        CALL    CVT1
        LD      DE,BLM
        CALL    CVT1
        LD      DE,EXM
        CALL    CVT1
        LD      DE,DSM
        CALL    CVT2
        LD      DE,DRM
        CALL    CVT2
        LD      DE,ALV0
        CALL    CVT1
        LD      DE,ALV1
        CALL    CVT1
        LD      DE,CKS
        CALL    CVT2
        LD      DE,OFF
        CALL    CVT2
;
        LD      HL,DSPLAY       ; dph table
        CALL    COSTR           ; print it all
        POP     HL
        LD      B,15
        CALL    HEXDMP          ; print hexdump of dpb
        CALL    COCRLF
        CALL    COCRLF
        RET
;
SKEW:   LD      A,ACROSS        ; number of records to display across screen

        LD      (NPRINT),A      ; save
        LD      A,(FCB1)        ; see if drive was specified
        DEC     A               ; test ff=no
        JP      P,REC2          ; yes, select drive
        CALL    FCN25           ; no, get current drive
REC2:   LD      E,A
        LD      C,A
        CALL    FCN14           ; select drive
        CALL    FCN31           ; get address of dph
        LD      E,(HL)
        INC     HL
        LD      D,(HL)
        EX      DE,HL
        LD      (NRECS),HL      ; save dph address for records per track
        LD      A,24            ; offset for seldsk:
        CALL    BIOS            ; hl=dph, de=xlate table
        LD      E,(HL)
        INC     HL
        LD      D,(HL)          ; de=dph
        PUSH    HL
        PUSH    DE
        PUSH    BC
        EX      DE,HL
        LD      (DPHADR),HL     ; and save
        CALL    HEXADR          ; display dph address
        LD      HL,DPH          ; trailer message for address
        CALL    COSTR           ; display
        POP     BC
        POP     DE
```

```
        POP     HL
        LD      A,E             ; get low byte
        OR      D               ; or with high
        JP      NZ,REC3         ; if zero no translate done
        LD      HL,NXLATE       ; no translate message
        CALL    COSTR
        RET
;
REC3:   LD      BC,0            ; clear record number
        XOR     A               ; clear a
        LD      (POSIT),A       ; initialize counter
;
RECLOP: LD      A,45            ; offset to sectran
        CALL    BIOS            ; hl=xlate (bc)
        CALL    PRINT           ; display results
        LD      HL,(NRECS)      ;
        DEC     HL
        LD      A,L
        OR      H
        JP      Z,SHOWD
        LD      (NRECS),HL
        INC     BC
        JP      RECLOP          ; till through
;
SHOWD:  LD      HL,TBLMSG       ; for skew table
        CALL    COSTR           ; display
        LD      A,20            ; number of entries
        LD      (POSIT),A       ; set counter
        LD      HL,(DPHADR)     ; get dph address
        LD      (LAST),HL       ; save it
        JP      ENTRY
;
MORE:   LD      HL,(LAST)       ; get last position
        INC     HL              ; update and
        LD      (LAST),HL       ; save it
ENTRY:  LD      E,(HL)          ; get byte
        LD      D,0             ; zero d
        LD      A,E             ; value in a
        CP      27              ; largest value
        JP      NC,ZERFIL       ;
        CP      1               ; start of table
        JP      Z,WHERE         ; what position
        CP      0               ;
        JP      NZ,NOPE         ; none of those
WHERE:  LD      A,(POSIT)       ; see where in table
        CP      20              ; first position of 20
        JP      NZ,ZERFIL       ; see if need zeros
NOPE:   EX      DE,HL           ; hl has value
        CALL    DECOUT          ; print decimal value
        LD      A,(POSIT)       ; get count
        CP      0               ; see if done
        RET     Z               ; yes, return
        LD      A,','           ; insert comma
        CALL    COUT
        CALL    COSPAC          ; insert space
        LD      A,(POSIT)       ; see if done
        CP      0
        JP      Z,ZERFIL        ; fill remainder
NOTDON: DEC     A               ; count down
        LD      (POSIT),A       ; and save
        JP      MORE
;
ZERFIL: LD      A,(POSIT)       ; how many zero's
        CP      0
        RET     Z               ; done if zero
        DEC     A               ; count down
        LD      (POSIT),A       ; and save
        LD      A,'0'
        CALL    COUT            ; print '0'
        LD      A,(POSIT)
        CP      0               ; see if through now
        RET     Z
        LD      A,','
        CALL    COUT            ; print ','
        CALL    COSPAC          ; print space
        JP      ZERFIL
;
; convert binary in hl to ascii decimal
;
DECOUT: LD      B,0             ; leading zero fill
        LD      DE,-100
        CALL    SUBTR           ; hundreds
        LD      DE,-10
        CALL    SUBTR           ; tens
        LD      A,L
        ADD     A,'0'           ; ascii bias
        JP      COUT            ; print and return to caller
;
; subtract powers of 10
;
SUBTR:  LD      C,'0'-1         ; ascii count
SUBT2:  INC     C
        ADD     HL,DE           ; add neg number
        JP      C,SUBT2
;
; one to many add one back
;
        LD      A,D             ; complement
        CPL                     ; de
        LD      D,A
        LD      A,E
        CPL
        LD      E,A
        INC     DE
        ADD     HL,DE
        LD      A,C             ; get count
;
; check for zero
;
        CP      '1'             ; less than 1?
        JP      NC,NZERO        ; no
        LD      A,B             ; check 0 flag
        OR      A
        LD      A,C             ; restore
        RET     Z               ; skip leading zero
        JP      COUT            ; print character
;
; set flag for non-zero character
;
NZERO:  LD      B,0FFH          ; set to zero
        JP      COUT            ; print
        RET
```

```
;
; print record entry
;
PRINT:  CALL    HEXADR          ; print hl as hex words
        CALL    COSPAC          ; insert space
        LD      A,(NPRINT)      ; see if end of line
        DEC     A
        LD      (NPRINT),A      ; update
        RET     NZ
        LD      A,ACROSS        ; how many lines across
        LD      (NPRINT),A      ; save
        CALL    COCRLF          ; start new line
        RET
;
; select disk, 'e' has drive number
;
FCN14:  PUSH    BC
        PUSH    DE
        PUSH    HL
        LD      C,14
        CALL    BDOS
        POP     HL
        POP     DE
        POP     BC
        RET
;
; get current disk number, returns current disk in 'a'
;
FCN25:  PUSH    BC
        PUSH    DE
        PUSH    HL
        LD      C,25            ; no drive given, get current disk
        CALL    BDOS
        POP     HL
        POP     DE
        POP     BC
        RET
;
; get disk parameters, returns 'hl'=dpb address
;
FCN31:  PUSH    BC
        PUSH    DE
        LD      C,31            ; hl = dpb for current disk
        CALL    BDOS
        POP     DE
        POP     BC
        RET
;
CVT2:   PUSH    HL              ; display 16 bit integer
        INC     HL
        CALL    CVT1
        EX      (SP),HL
        CALL    CVT1
        POP     HL
        RET
;
CVT1:   LD      A,(HL)          ; display byte at 'hl'
        INC     HL
CVT1A:  CALL    HEXBYT          ; a,c=ascii display
        LD      (DE),A
        INC     DE
        LD      A,C
        LD      (DE),A
        INC     DE
        RET
;
; Subroutine to do an addressed dump of one line. HL point to data,
; 'B' has length
;
HEXDMP: CALL    HEXADR          ; display hl contents in hex
        LD      A,':'
        CALL    COUT
        CALL    COSPAC
;
HEXLIN: PUSH    BC
        PUSH    HL
HEXL2:  LD      A,(HL)
        CALL    COHEX           ; display hl in hex
        CALL    COSPAC          ; space
        INC     HL

        DEC     B
        JP      NZ,HEXL2
        POP     HL
        POP     BC
        RET
;
; Subroutine to print hl as an address in hex
;
HEXADR: LD      A,H
        CALL    COHEX
        LD      A,L
        CALL    COHEX
        RET
;
; print the hex byte in 'a'
;
COHEX:  PUSH    AF              ; save byte
        CALL    HEXLFT          ; get left half
        CALL    COUT            ; and print
        POP     AF              ; restore byte
        CALL    HEXRHT          ; right half and
        CALL    COUT            ; print
        RET
;
HEXLFT: RRA                     ; make left nibble
        RRA
        RRA
        RRA
;
HEXRHT: AND     0FH             ; make right nibble
        CP      0AH
        JP      C,HEXLR
        ADD     A,'A'-3AH
;
HEXLR:  ADD     A,'0'
        RET
;
; suboutrine to return the hex value of 'a',
; least significant half in 'c', most significant in 'a'.
; used when ascii-hex is to be stored.
;
```

```
        POP     AF
        RET
;
; subroutine to write bytes addressed by hl, terminated by last byte with
; 80 bit set.
;
COSTR:  LD      A,(HL)          ; get byte
        CALL    COUT            ; print
        LD      A,(HL)          ; again and test
        RLCA                    ; for bit 7 set
        RET     C               ; yes, return
        INC     HL              ; next byte
        JP      COSTR           ; and continue
;
; bios service, offset in 'a'
;
BIOS:   PUSH    BC
        PUSH    DE
        LD      HL,(BOOT+1)     ; get warm boot address
        ADD     A,L             ; add offset to wboot
        LD      L,A             ; to address of service
        PUSH    HL              ; save it
        LD      HL,BIORET       ; return address
        EX      (SP),HL
        JP      (HL)
;
BIORET: POP     DE
        POP     BC
        RET
;
;  message strings
;
DSPLAY:
        DEFB    CR,LF,'DISK PARAMETER BLOCK for drive '
DRIVE:  DEFS    1
        DEFB    ':',CR,LF
        DEFB    'SPT:  '
SPT:    DEFS    4
        DEFB    'H',TAB,'Records Per Track.',CR,LF
        DEFB    'BSH:  '
BSH:    DEFS    2
        DEFB    'H',TAB,'Block Shift Factor',CR,LF
        DEFB    'BLM:  '
BLM:    DEFS    2
        DEFB    'H',TAB,'Block Mask',CR,LF
        DEFB    'EXM:  '
EXM:    DEFS    2
        DEFB    'H',TAB,'Extent Mask',CR,LF
        DEFB    'DSM:  '
DSM:    DEFS    4
        DEFB    'H',TAB,'Disk Size-1',CR,LF
        DEFB    'DRM:  '
DRM:    DEFS    4
        DEFB    'H',TAB,'Directory Size-1',CR,LF
        DEFB    'ASV0: '
ALV0:   DEFS    2
        DEFB    'H',TAB,'Allocation Vector',CR,LF
        DEFB    'ALV1: '
ALV1:   DEFS    2
        DEFB    'H',TAB,CR,LF
        DEFB    'CKS:  '
CKS:    DEFS    4
        DEFB    'H',TAB,'Directory Check size',CR,LF
        DEFB    'OFF:  '
OFF:    DEFS    4
        DEFB    'H',TAB,'Offset (Number of reserved tracks)',CR,LF,LF
;
DMPMSG: DEFB    '"DPB address and hex dump of table"'
        DEFB    CR,LF+80H
;
NXLATE: DEFB    BELL,'Diskette needs no translation',CR,LF+80H
;
TBLMSG: DEFB    'Skew table decimal values',CR,LF+80H
;
DPH:    DEFB    ': (DPH address)',CR,LF+80H
; User help
;
HEXBYT: PUSH    AF              ; save
        CALL    HEXRHT          ; get right nibble
        LD      C,A             ; save in c
        POP     AF              ; restore and get
        CALL    HEXLFT          ; left nibble
        RET
;
; subroutine to write 'a' to console
;
COUT:   PUSH    BC
        PUSH    DE
        PUSH    HL
        AND     7FH             ; strip bit 7
        LD      E,A
        LD      C,2             ; cp/m fcn 2
        CALL    BDOS            ;
        POP     HL
        POP     DE
        POP     BC
        RET
;
; print cr,lf to the console
;
COCRLF: PUSH    AF
        LD      A,CR            ; carriage return
        CALL    COUT            ; print it
        LD      A,LF            ; line feed
        CALL    COUT            ; and print it
        POP     AF
        RET
;
; subroutine to write a space to the console
;
COSPAC: PUSH    AF
        LD      A,SPC           ; space
        CALL    COUT            ; and print it

HLPMSG: DEFB    'DISKPRAM.COM - <H E L P>',CR,LF,LF
        DEFB    'Syntax:',CR,LF
        DEFB    'A0>DISKPRAM <cr>     - DPB and SKEW table for defalt drive'
        DEFB    CR,LF,'A0>DISKPRAM d:       - Tables for selected drive'
        DEFB    CR,LF,'A0>DISKPRAM ? or /   - Displays this message'
        DEFB    CR,LF
;
```

```
        DEFB    CR,LF,'This program is designed to provide the user with a'
        DEFB    ' tool'
        DEFB    CR,LF,'that is able to display the DPB and SKEW table of a'
        DEFB    ' CP/M'
        DEFB    CR,LF,'2.2 system.  This information can be used as the'
        DEFB    ' decimal'
        DEFB    CR,LF,'input for a program such as ESET.COM written for the'
        DEFB    ' AMPRO'
        DEFB    CR,LF,'system.  With ESET the AMPRO''s E: drive may be used to
        DEFB    ' read'
        DEFB    CR,LF,'and write alien 5" diskette formats',CR,LF,LF+80H
;
NRECS:  DEFS    2
NPRINT: DEFS    1
DPHADR: DEFS    2
POSIT:  DEFS    1
LAST:   DEFS    2
;
        DEFS    30
STACK:  EQU     $
;
        END
```

DISKPRAM, show in Listing 2, is a public domain tool which is designed to provide us with the basic information we will require. What is needed is to run DISKPRAM on the alien system, and make a record of the parameters listed. The values returned by this tool may be installed in the "E" disk DPB. The disk type byte information will generally be known. Those portions of the type byte data which are not known may be determined by some experimentation with the available parameters. The menu driven, public domain utility "ESET.COM" may be used to temporarily redefine the "E" disk parameters. This program is described in the AMPRO technical manual.

(Continued on page 51)

TCJ ORDER FORM

Subscriptions		U.S.	Canada	Surface Foreign	Total
6 issues per year ☐ New ☐ Renewal	1 year	$16.00	$22.00	$24.00	
	2 years	$28.00	$42.00		
Back Issues		$3.50 ea.	$3.50 ea.	$4.75 ea.	
Six or more #'s		$3.00 ea.	$3.00 ea	$4.25 ea.	
				Total Enclosed	

All funds must be in U.S. dollars on a U.S. bank.

☐ Check enclosed ☐ VISA ☐ MasterCard Card # _____

Expiration date_____ Signature _____

Name _____

Address _____

City_____State_____ZIP _____

The Computer Journal

190 Sullivan Crossroad, Columbia Falls, MT 59912 Phone (406) 257-9119

*** **Orders can also be entered by modem on the TCJ BBS (406) 752-1038 by leaving a private message for SYSOP with the required information.**

Feedback

(Continued from page 35)

vironmental monitor. All data will be collected as frequencies (counts), so the sixteen bits will provide a convenient input mode. This is primarily because my moisture detector (the most difficult) is a frequency dependent device. Then resistance, light and events will just be counted, and entered as numbers also, to be decoded by software.

Yes, software — I have none. Some modifications have been made to the board to make it more transportable to other beginners. The memory is now a 6116 2K×8 RAM, which exactly matches a 2716 EPROM and a 52B13 ($1.95 Jameco) EEPROM. (See Byte February, 1984, pages 343-344). The software (which does not exist — yet!) is to be completely dumped from a CP/M machine over the parallel or serial ports. This is to be a ROMless computer. If the builder/user does not have a EPROM Programmer handy, the entire program could be dumped from disk into the RAM or EEPROM by way of a full computer.

Thus far, the needs for the software areas follows: Warm boot and reset commands to initialize the system. There should be some user commands to set up the ports for data collection. These will be in English (on the screen) but the responses will cause the host computer to send appropriate machine language instructions to the Z-80 board. These will be in the form of; "Port 0 is to be examined every : [A] MINUTE; [B] HOUR ; [C] DAY." Answering this question sets the timer function to collect data from port 0 at the appropriate interval, and store it in RAM memory. The menu would also have items like "To return data enter D [RETURN]". At this time the small board would dump the RAM data into the host.

Using the EEPROM makes the operating system non-volatile and easily upgraded. The entire assembly process could be supplied on a disk. I have the wire list and component layout for wire wrap in a WordStar format now. I don't see why it could not be put in a CP/M file like README.TYP. Anyone building this should buy the book. Some of the items (such as the video and casette storage) are obsolete, but he has some very good discussions on the CPU and programming, as well as the operating theory of computers in general.

My soldering iron is warm now. If this project works, I will let you know. If anyone is interested in developing software of the type described above, the assistance would be appreciated. The profitability of this system is close to absolute zero — that's why it should be in the public domain.

N.E.

Editor's Note: The documentation will be very much appreciated for our reference files which will be used to assist people working with older systems. Can anyone help with the info N.E. needs?

His project sounds interesting and if enough readers express an interest we'll try to get him to submit an article, in the meantime if someone is willing to assist him with the software, contact TCJ for his address.

Echelon Responds

On page 46 of TCJ issue 24 appears a letter from "D.D." Item (4) of the letter states "The Z-System is public domain, but surely the BIOS from Micromint or Ampro is not public domain; specifically, which files is one allowed to copy for friends?"

Our reply. First, Z-System is NOT public domain, period. All aspects of ZCPR3 and ZRDOS that make up the Z-System are copyrighted. But ZCPR3, CP/M CCP replacement, is special. Programs and utilities written by Richard Conn can be freely copied and given to friends, but they can not be sold. You need a license from Echelon to sell ZCPR3 programs

Second, ZRDOS, the CP/M BDOS replacement combined with ZCPR3 is Z-System, is licensed and supported by Echelon. You need a license from Echelon to sell ZRDOS. And it is against the law to copy is and give it to friends. Subject files are ZRDINSnn.COM and ZRDOSnn.BIN, where "nn" is the version number. But support utilities associated with ZRDOS can be feeely given to friends who have obtained ZRDOS package from us. These are AC, VTYPE, VIEW, SFA, DFA, COMP, DOSERR, DOSVER, DISKRST, LOGGED, DRO, SRO, and SRW. Each was written to take advantage of features of Z-System and thus only run if ZRDOS is present.

Z-System is fully copyrighted by Richard Conn, Denis Wright, and Echelon. To sell and support Z-System is one reason Echelon is in business.

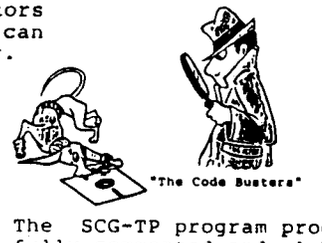
Last, those who receive Z-System fully installed on their computer, e.g., on Micromint's SB180, and want to become involved with the Z-System community, should subscribe to our fortnightly newsletter, Z-NEWS. It sells for $24.00 per year. Also we have a free catalog of interesting Z80 and HD64180 Z-System and CP/M compatible software products. Additionally, we support over 60 remote access systems called Z-Nodes around the world acting as electronic forums for people interested in Z-System.

Frank Gaude'
President
Echelon, Inc.

Ever Wondered What Makes *TURBOPASCAL* Tick?

Source Code Generators by C. C. Software can give you the answer.

"The darndest thing I ever did see..."
"... if you're at all interested in what's going on in your system, it's worth it."
Jerry Pournelle, BYTE, Sept '83



MasterCard VISA

The SCG-TP program produces fully commented and labeled source code for your TURBO-Pascal system. To modify, just edit and assemble. Version 3.00A (Z80) is $45. SCG's available for CP/M 2.2 ($45) and CP/M+ ($75). Please include $1.50 postage (in Calif add 6.5%).

C. C. Software, 1907 Alvarado Ave. Dept M
Walnut Creek, CA 94596 (415)939-8153

CP/M is a registered trademark of Digital Research, Inc.
TURBO Pascal is a trademark of Borland International

Computer Corner

(Continued from page 52)

cheap), it is standardized, plenty of suppliers, biggest assortment of cards and CPUs. I think the biggest thing going for the STD bus is the manufacturer's innovative abilities. There are several manufactureers who have complete systems with 3 inch disks all in a standard cage running compatible PC operations.

The most important aspect is support and STD bus people have been supporting other manufacturers for some time and know just what they need. One company I have had some experience with is Giddings and Lewis they make big lathes as well as the electronics to drive them. We happen to have their old style systems but they also make STD products, or I should say they buy most of their cards (CPU and memory) except the special I/O they need. I attended one of their sales seminars on the STD unit (also a factory school on the big system) and was given not only a demonstration of how the system worked but all their source code for the special math routines they use. They feel that if you are going to properly use the product (or maintain it) you must have all the code they use. When asked about who supplies the cards which they don't make, they gladly tell you where to get them. Their attitude is that they sell a complete service of large machines and controllers and your satisfaction with their entire line is more important than a few lines of code they wrote.

On the other side of the coin is a product which I personally was able to buy recently. The same friends who sold me my portable, also had a GIMIX system they got at auction which was gathering more dust than use. I removed this unit from their warehouse and have been trying to use it ever since. What really got me was the factory response when I called them for information. Now their literature says they do not make any consumer products, only industrial use is warranted, but if my use had been industrial I certainly would not use them. What happened was nothing, they stated they destroyed all information on older products and would not sell me anything to help me bring the unit up. When I asked what I should do they said to use it for parts. I was able to use a third party to receive information and have since found out that these units are their current versions but with them destroying their documentation it is rather hard to know what is really current. I have several disk cards to sell as this unit was a disk drive tester with eight disk cards and must have cost someone over $9,000 in the beginning. If I had been a manufacturer and spent nine grand for several systems, only to have the manufacturer say I must buy a whole new system to fix my old one, that would be the last time I did business with him.

Well that one call is the last time I will call GIMIX and they are not the only company I will refuse to do business with due to poor support or documentation. Another big company which I will have little to do with is GE. We have one of their big industrial controllers and it must be the most difficult unit to work on. Becuse I had had so much trouble figuring out their documentation and many other forms of troubles with them, when it became time to upgrade an old lathe I got our managers to buy a older style Gidding and Lewis (G&L) controller. The managers wanted to buy another GE controller because of the name, until I explained all the trouble I had, then they knew why they hadn't heard the G&L name much before this, the product caused little problems. The GE was a mess and they lost $300,000 to G&L. You might say that was only one case, not so, we are buying our second wrapping machine from another company because the first company's software people and support engineers were unorganized and too tight with their software. This unit used a G&L controller but they couldn't program it properly, so we went with a company that was open and using a multibus system running multiple CPUs. The first company's poor manament structure cost them two wrapping machine at over $300,000 each. So when people talk to me about how important support is, I say very.

Trying to pull all this together and conclude this column is not an easy task. Art's editorial last issue got me thinking of what type of products to use and how the industry has gone hardware crazy. I pointed out how support is important and keeping in mind the basic functions you hope to achieve with your system. I have many systems at home and work to play with, but still find my old CP/M systems running Wordstar to be the best. Why, because they are easy to use and simple to run. I can learn all the insides without too much problem, and do all my own work without spending a lifetime of study. For me SIMPLE is still the best.

Things to keep in mind are your objectives, costs, support, and your skills level. Programs that do ten times more than you need may do what you want so poorly that a simpler and cheaper product would be more appropriate (which is why I like WordStar). Hardware systems that have complex operating systems may leave less than 64K for your programs (PCDOS running Framework and 384K mem). Graphic systems are nice but if they are so complex to program that you will not be able to write to them yourself, why bother?

A solution for some user may be the same one I have chosen. Some time ago I started on a 68K FORTH operating system. Hawthorne Technology has saved me from that task (at present) but I have become stronger in my feelings that some form of a standard language and operating system is needed. I can use four different CPUs (6809, Z80/8080, 8088, 68K) and need some common way of going between them freely. FORTH is currently the only option for me. STD Bus users are also using FORTH because it is fast and can work in small memory systems. If you're interested in FORTH look at past FORTH articles as well as my TUTOR program elsewhere in TCJ.

I think I will save some comments for next month, so happy computing. ■

Disk Parameters

(Continued from page 49)

Well, we have more theory than application in this article,and because of the size of the listings and figures we had better bring this session to a halt. I suggest you study the data given in this issue until the meaning of each byte of these parameters is understood fully. It would not hurt to design a disk format of your own, in practice, to better understand the decisions made by the designer of the AMPRO format. ■

The public domain DISKPRAM program source code and object code files plus other related files are available on eight bit 5.25" AMPRO, Kaypro, Morrow, and several other popular formats for $10 postpaid in the U.S.

■ ■ ■

THE COMPUTER CORNER

A Column by Bill Kibler

Well here I sit before my new computer on Thanksgiving Day putting down my thoughts before I forget them. You might think I am at my home, but no I am at my mother's waiting for the rest of the family to arrive. The machine I am using is my new Heath/Zenith 171® portable PC. This is the same unit the IRS bought and yea the same one that Morrow sold to ZDS for a small sum of money, without any royalities. I needed the machine for my college work and it is my only PC compatible machine (and will be the only one too). Thanks to some friends of mine I was able to pick it up for much less than the going price, but even then the unit is a bit expensive for what you get. I have found the keyboard to be very good, in fact I like it better than any PC or PC clone's keyboard I have tried. After getting this I noticed how far apart most of the other keyboards keys were (PC types). What all this is leading into is my increased use and knowledge of PC programs.

My masters work is moving along well and most of the work I am doing now is on PCs with PC type programs. What I have found out so far is the current trend in PC programs — use of the function and cursor keys exclusively. I must admit that my first wordprocessing program was WordStar, mostly because I worked there. Now I still use this program and am writing this article on it. I have tried several other PC word processors and have not liked a single one of them. The reason is their use of the function keys. WordStar® can use function keys if you want, but most of the cursor movement can be made without lifting your hands from the home postion. I recently spent a month on Framework® , an integrated package of wordprocessing, database, and spreadsheet. This program has some good features but for the most part I hated it. It seemed every third keystroke was a function key or cursor movement key, without any optional ways around them. What really made me hate the program was their use of the + and - cursor position keys. If you look at the key codes produced by a PC keyboard there are different key codes for every possible use of the keys. The number pad can be numbers, cursor positioning functions, as well as control and alternate options too. What Framework and other programs do is use all these possible combinations to get their options.

Wordstar divided their options into groups and you get to those groups by doing a control key entry, then use a regular key to run the option. These other prorams are using a single keycode for an option. This means you must have a compatible keyboard and you must also (in most cases) move your hands from the home keys to use or invoke an option. I will admit that any user will probably always like their first system and first wordprocessor, but these new unit didn't really look at the old unit and build on their good or bad options. Some of these new units handle text better than Wordstar for desktop publishing options (like multicolumn formating), but overall I feel the programmers got impressed by the options available and went crazy. This going crazy with options and technology is really what I want to talk about.

I just finished reading the last issue of Computer Journal and feel that what Art was trying to say and what I just talked about are related. In the early days, computers did very little and anything that worked was wonderful. Most of us bought systems just to see what could be done. That time is gone now, althought I still buy systems to check out (more later), but I basically buy them to use for some productive work. When doing wordprocesing (which is approximately 70% of all computer use) I want something which allows me to write easily and quickly without a lot of hand movement. Now all these extra bells and whistles are nice as long as they don't stop me from doing the original objective, writing. I have found that statement to be true also for operating systems as well as computer languages. I think anybody who has used Turbo Pascal® to write a program will say that one of its best features is the ease with which you can use it. From the Wordstar like editor to the error correction process, I find that with Turbo I can do in four hours what would have taken two days using conventional systems. The why is the objective approach of the system.

I currently feel that the computer industry is running amuck with all the new found speed and options. Programs (and hardware too) are being pushed to their limits, in many cases just to find those limits. Now I have little complaint about finding out just how far you can push things, I did that when I first learned how to drive. After finding myself in a ditch one night, I learned when to push and when not to. The industry hasn't learned when not to push, and I feel this is due to sell, sell, and more sell. The competion is fierce and only getting worse. Anybody who really says "I liked it when...", will be considered old fashioned or not with it by most computer users. Nice ideas, but what really are we trying to do, I feel the objective is to get the job done in the most cost effective and time efficent manner. Computers can do that but only if used properly.

I think Art and others like him are heading in the right direction by using combinations of existing products to achieve their goals. One area that I have been pushing for some time is the use of the STD bus for industrial control. Any company that started using STD bus with Z80 or 6809 CPUs can now upgrade to 8088 or 68000 by only changing their CPU cards. All their I/O and memory will most likely work without any changes. This type of upgrading is hard to beat. Where I work we have many industrial controllers all with different software and hardware. To maintain or program these systems requires sending the personnel to a factory school (always on the other side of the USA) and then being locked in to their price and service. STD bus products are available in many configurations with some local dealer or manufacturer support available. Because the basic CPU can be the same for all uses, only learning one system is needed for programming or maintaining. When it comes to really special uses where you must build your own, with STD you only build that part which can't be found elsewhere. The reason I like STD bus over others is cost (they're

(Continued on page 51)

Back Issues Available:

Issue Number 1:
- RS-232 Interface Part One
- Telecomputing with the Apple II
- Beginner's Column: Getting Started
- Build an "Epram"

Issue Number 2:
- File Transfer Programs for CP/M
- RS-232 Interface Part Two
- Build Hardware Print Spooler: Part 1
- Review of Floppy Disk Formats
- Sending Morse Code with an Apple II
- Beginner's Column: Basic Concepts and Formulas

Issue Number 3:
- Add an 8087 Math Chip to Your Dual Processor Board
- Build an A/D Converter for the Apple II
- Modems for Micros
- The CP/M Operating System
- Build Hardware Print Spooler: Part 2

Issue Number 4:
- Optronics, Part 1: Detecting, Generating, and Using Light in Electronics
- Multi-User: An Introduction
- Making the CP/M User Function More Useful
- Build Hardware Print Spooler: Part 3
- Beginner's Column: Power Supply Design

Issue Number 5:
- Optronics, Part 2: Practical Applications
- Multi-Processor Systems
- True RMS Measurements
- Gemini-10X: Modifications to Allow both Serial and Parallel Operation

Issue Number 6:
- Build High Resolution S-100 Graphics Board: Part 1
- System Integration, Part 1: Selecting System Components
- Optronics, Part 3: Fiber Optics
- Controlling DC Motors
- Multi-User: Local Area Networks
- DC Motor Applications

Issue Number 8:
- Build VIC-20 EPROM Programmer
- Multi-User: CP/Net
- Build High Resolution S-100 Graphics Board: Part 3
- System Integration, Part 3: CP/M 3.0
- Linear Optimization with Micros

Issue Number 9:
- Threaded Interpretive Language, Part 1: Introduction and Elementary Routines
- Interfacing Tips & Troubles: DC to DC Converters
- Multi-User: C-NET
- Reading PCDOS Diskettes with the Morrow Micro Decision
- DOS Wars
- Build a Code Photoreader

Issue Number 14:
- Hardware Tricks
- Controlling the Hayes Micromodem II from Assembly Language, Part 1
- S-100 8 to 16 Bit RAM Conversion
- Time-Frequency Domain Analysis
- BASE: Part Two
- Interfacing Tips and Troubles: Interfacing the Sinclair Computers, Part 2

Issue Number 15:
- Interfacing the 6522 to the Apple II
- Interfacing Tips & Troubles: Building a Poor-Man's Logic Analyzer
- Controlling the Hayes Micromodem II From Assembly Language, Part 2
- The State of the Industry
- Lowering Power Consumption in 8" Floppy Disk Drives
- BASE: Part Three

Issue Number 16:
- Debugging 8087 Code
- Using the Apple Game Port
- BASE: Part Four
- Using the S-100 Bus and the 68008 CPU
- Interfacing Tips & Troubles: Build a "Jellybean" Logic-to-RS232 Converter

Issue Number 17:
- Poor Man's Distributed Processing
- BASE: Part Five
- FAX-64: Facsimile Pictures on a Micro
- The Computer Corner

Interfacing Tips & Troubles: Memory Mapped I/O on the ZX81

Issue Number 18:
- Parallel Interface for Apple II Game Port
- The Hacker's MAC: A Letter from Lee Felsenstein
- S-100 Graphics Screen Dump
- The LS-100 Disk Simulator Kit
- BASE: Part Six
- Interfacing Tips & Troubles: Communicating with Telephone Tone Control, Part 1
- The Computer Corner

Issue Number 19:
- Using The Extensibility of Forth
- Extended CBIOS
- A $500 Superbrain Computer
- BASE: Part Seven
- Interfacing Tips & Troubles: Communicating with Telephone Tone Control, Part 2
- Multitasking and Windows with CP/M: A Review of MTBASIC
- The Computer Corner

Issue Number 20:
- Designing an 8035 SBC
- Using Apple Graphics from CP/M: Turbo Pascal Controls Apple Graphics
- Soldering and Other Strange Tales
- Build a S-100 Floppy Disk Controller: WD2797 Controller for CP/M 68K
- The Computer Corner

Issue Number 21:
- Extending Turbo Pascal: Customize with Procedures and Functions
- Unsoldering: The Arcane Art
- Analog Data Acquisition and Control: Connecting Your Computer to the Real World
- Programming the 8035 SBC
- The Computer Corner

Issue Number 22:
- NEW-DOS: Write Your Own Operating System
- Variability in the BDS C Standard Library
- The SCSI Interface: Introductory Column
- Using Turbo Pascal ISAM Files
- The AMPRO Little Board Column
- The Computer Corner

Issue Number 23:
- C Column: Flow Control & Program Structure
- The Z Column: Getting Started with Directories & User Areas
- The SCSI Interface: Introduction to SCSI
- NEW-DOS: The Console Command Processor
- Editing The CP/M Operating System
- INDEXER: Turbo Pascal Program to Create Index
- The AMPRO Little Board Column
- The Computer Corner

Issue Number 24:
- Selecting and Building a System
- The SCSI Interface: SCSI Command Protocol
- Introduction to Assembly Code for CP/M
- The C Column: Software Text Filters
- AMPRO 186 Column: Installing MS-DOS Software
- The Z Column
- NEW-DOS: The CCP Internal Commands
- ZTIME-1: A Realtime Clock for the AMPRO Z-80 Little Board

Issue Number 25:
- Repairing & Modifying Printed Circuits
- Z-Com vs Hacker Version of Z-System
- Exploring Single Linked Lists in C
- Adding Serial Port to Ampro Little Board
- Building a SCSI Adapter
- New-DOS: CCP Internal Commands
- Ampro '186: Networking with SuperDUO
- ZSIG Column

Issue Number 26:
- Bus Systems: Selecting a System Bus
- Using the SB180 Real Time Clock
- The SCSI Interface: Software for the SCSI Adapter
- Inside AMPRO Computers
- NEW-DOS: The CCP Commands Continued
- ZSIG Corner
- Affordable C Compilers
- Concurrent Multitasking: A Review of DoubleDOS
- The Computer Corner